The Java Query Language

by

Darren Willis

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2008

Abstract

This thesis describes JQL, an extension to Java which provides object querying. Object querying is an abstraction of operations over collections, including operations that combine multiple collections, which would otherwise have to be manually implemented. Such manual implementations are 'low-level'; they force developers to specify how an operation is done, rather than what the operation to do is. Many operations over collections can easily be expressed as queries. JQL provides a Java-like syntax for expressing these queries, an optimizing query evaluator that can dynamically reconfigure query evaluation, and a caching system that allows querying to replace common collection operations with incrementally cached versions.

Acknowledgments

Special thanks to David J. Pearce and James Noble for their supervision; other people to thank for their input include but are not limited to Gavin Bierman, Alisdair Wren, Anna Ho, Erik Meijer, Helen Breeze, Emi Kakinuma, Mikael Brockman, Melissa Gyetse, Sam Stephenson, the ELVIS people, and a bunch of others. This work was produced with the support of the Royal Society of New Zealand Marsden Fund.

Contents

Chapter 1

Introduction

Abstraction is the cornerstone of modern programming languages. The flexibility, ease, and portability offered by writing programs at higher and higher levels of abstraction is well understood [12]. Object-Oriented Programming (OOP) is one of the most successful techniques for abstraction. Bundling together objects into collections of objects, and then operating on these collections, is a fundamental part of mainstream object-oriented programming languages.

Implementing operations over these collections with conventional techniques severely lacks in abstraction. Step-by-step instructions must be provided as to how to iterate over the collection, select elements, and operate on the elements. Manually specifying the implementation of these operations fixes how they are to be evaluated, rendering it impossible to accommodate changes in the state of the program that could make another approach superior. This is especially problematic when combining two collections of objects together, frequently an expensive operation.

Object querying provides a simple abstraction for operations over the elements in a collection, or for operations that combine collections. Object querying is a way to select an object, or set of objects, from a collection or set of collections. This selection is based on criteria the developer specifies, such as the value of certain fields, etc. Using queries in place of a manual

implementation frees developers from implementing the operation themselves.

The additional abstraction offered by object querying also allows for easy caching of query results. Using incrementalization techniques, which maintain the cached query results in-line with changes in the program, querying becomes a practical backend for implementing an abstract data type that self-optimizes for its actual usage.

In this thesis, we make the following contributions:

- We present JQL, the Java Query Language, which provides an object querying system for Java, to allow programmers to express complex queries declaratively.
- We describe JQL's incrementalized query caching system, which caches query results and maintains these cached results to be accurate while the program executes.
- We present experimental data comparing JQL's performance to manual implementations of equivalent queries, showing the performance of JQL's caching system.
- We present a small study into the use of loops within existing Java programs, investigating what loops are usually used for.

This thesis is organized as follows:
Chapter 2 presents a study of loops in Java programs, and introduces JQL.

Chapter 3 describes the JQL language and the prototype implementation.

Chapter 4 details the incrementalized query caching system added to JQL.

Chapter 5 has related work.

Chapter 6 provides a summary, and postulates future work.

Chapter 2

Loops and Queries

Loops are one of the fundamental control structures in programming languages with imperative roots. It has been noted, at least as far back as 1978, that "the majority of loops are concerned with sequential processing of data structures"[41]. In this chapter we present a small study into the actual usage of loop constructs in real-world Java programs, give an overview of how object querying matches up to these usages, and describe the benefits and drawbacks of using object querying instead of loops.

2.1 Operations Over Collections

Object oriented programs need to implement operations that manipulate all elements in a collection. This is conventionally done using an iterating loop, sometimes containing several `if`s and `break`s. To see how applicable querying is for abstracting loop operations over collections, we have investigated actual loop operations in a small selection of Java programs. These programs were selected at random from Sourceforge, a popular online repository of open source software projects. Table 2.1 has some details of these programs. We found that the majority of loops can be categorized into one of a few broad groups of collection manipulators.

The categories we have used are:

Name	Version	LOC
Robocode (Game)	1.2.1A	23K
RSSOwl (RSS Reader)	1.2.3	46K
ZK (AJAX Framework)	2.2.0	45K
Chungles (File Transfer)	0.3	4K
Freemind (Diagram Tool)	0.8	70K
SoapUI (WebService Testing)	1.7b2	68K

Table 2.1: Java Benchmark Programs

- **Map** — apply a given function to every element of a collection
- **Filter** — construct a sub-collection from a collection using a selection predicate.
- **Product** — apply some function or filter to the product of two (or more) collections.
- **Reduce** — produce a single value from a collection (e.g. a sum).
- **Other** — Collection operations that do not fit readily into the other categories (e.g. sort, reverse, etc.).
- **Non-Collection** — loop operations that do not operate over a collection.

We now discuss these in more detail.

Map.

Maps simply apply statements to every element in a collection. Some maps return a new collection built out of the return values for these operations. Maps can already be implemented simply, using Java's 'enhanced' `for` loop construct.

```
for(Robot r : robots) {
  if(!r.isDead()) { r.method(); }
}
```

Figure 2.1: A simple example of a filter operation.

Filter.

Filter operations select elements out of collections. For example, the Robocode game frequently operates on all `Robot` objects in the main list of `Robots`, except those already destroyed. A simplified view of how this appears in the game's code is given in Figure 2.1.

Filtering operations frequently have several different filters, with different operations applied to different sets of objects, depending on which filter expressions they pass. These operations are usually expressed as either a series of `if-else` or `switch` statements within a loop. For example, in Chungles, a program for sharing files over a local network, the UI generates a list of shared items for a user and shows a different icon for folders than for files. A slightly simplified version of the code is given in Figure 2.2.

Product.

These operations are 'joins' between collections; they combine two collections together, and are, generally, the most expensive loop operations. These represent the most costly collection manipulations, and have the most to gain from query abstractions. Dynamic re-configuration of these operations (which we outline in Section 2.4) and query caching can greatly speed up these operations. A (simplified) example from Robocode, which increments a robot's 'survival' score for every dead robot not on that robot's team, is given in Figure 2.3.

```
for(String s : names)
 item = new TreeItem();
 item.setText(s.substring(1));
 switch(s.charAt(0)){
  case('D'){
    item.setImage(new Image("images/folder.gif");
    break;}
  case('F'){
    item.setImage(new Image("images/file.gif");
    break;}
}}
```

Figure 2.2: An example of an operation with multiple filters — one filtering out elements that start with 'D' and one filtering those that start with 'F'.

```
for(Robot r : robots) {
  if(!r.isDead()) {
    for(Robot dead : deadRobots) {
      if(r.team == null || r.team != dead.team) {
        r.scoreSurvival();
} } } }
```

Figure 2.3: An example product operation, that operates over `robots` and `deadRobots`.

Reduce.

The `Reduce` category consists of operations which reduce a collection(s) to either a single value or a very small set of values. Summing the elements of a collection is perhaps the most common example of this. Concatenating a collection into one large string is another.

Operation	Robocode	RSSOwl	ZK	Chungles	Freemind	SoapUI
Map	38	117	140	24	211	372
Filter	92	109	124	18	160	154
Product	8	2	4	0	2	1
Reduce	21	34	14	6	30	39
Other	6	27	22	0	15	34
Non-Collection	61	38	35	22	674	92
Total	**226**	**327**	**339**	**70**	**1092**	**692**

Table 2.2: Categorizations of loop operations in Java programs. Note that " Filter" and "Product" are the two loop classes which can benefit from the incrementalized query cache approach outlined in this paper.

Other.

Loops classified as `Other` are mostly loops on collections which either: depend on or change the position of elements in the collection, or operate on more than one element of the collection at a time.

Non-Collection.

These loops are those which are unrelated to collections — loops for reading from a file, or sleeping in a thread, for example.

Using this taxonomy we examined, by hand, a small set of open source Java programs (these are detailed in Table 2.1). The results of our analysis are presented in Table 2.2 and Figure 2.4. Except for the unusual case of Freemind (some 644 of Freemind's 'Non-Collection' loops appear to be generated code), the majority of loops are operations over collections; furthermore most of these loops fall into a small range of categories. We now outline object querying, an abstraction that correlates with several of these categories.

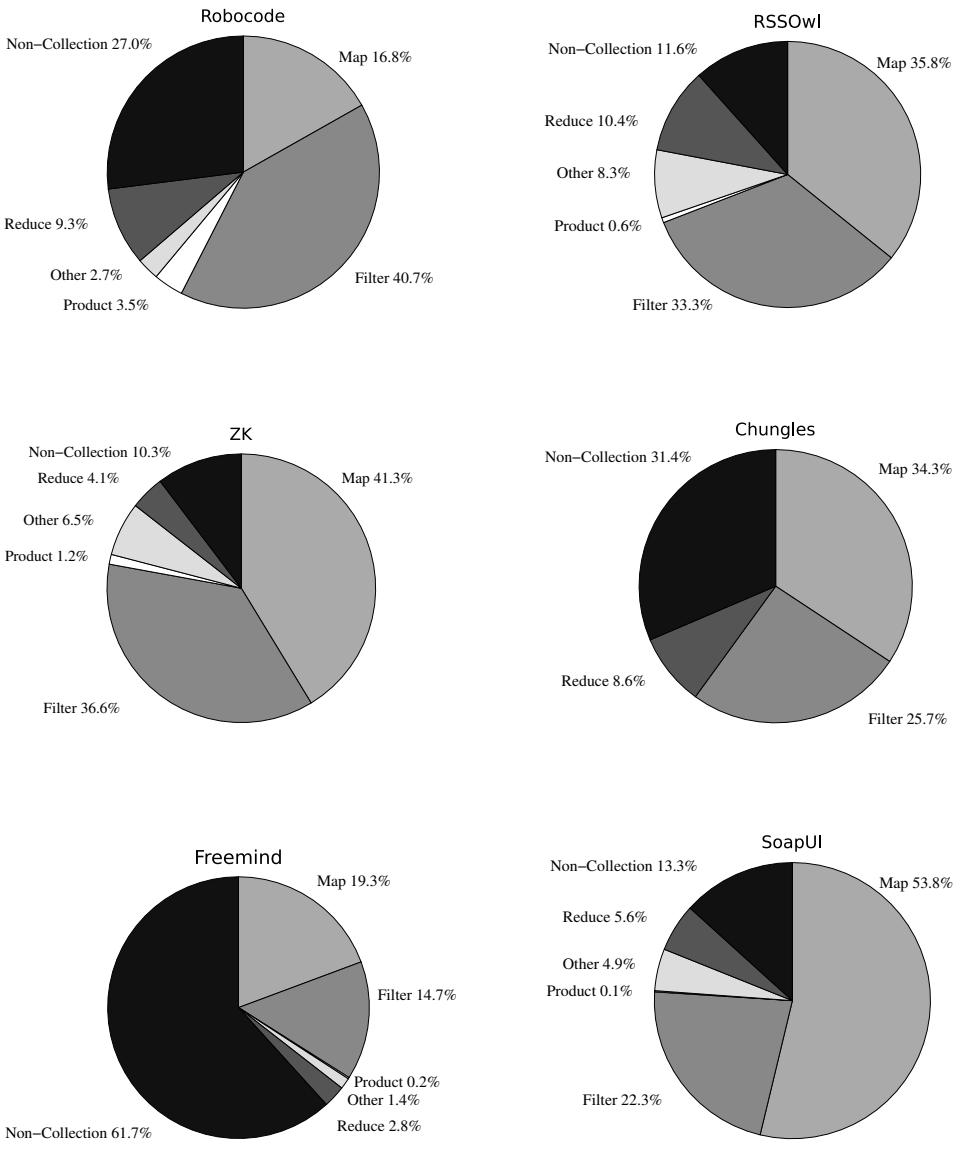


Figure 2.4: Graphical presentation of the data from Table 2.2.

2.2 Object Querying

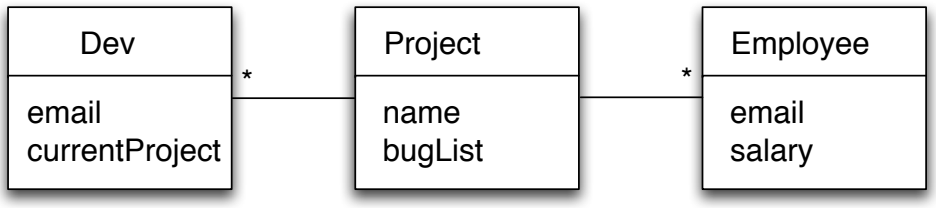
Object querying is a means for selecting objects out of collections of objects. Queries can be evaluated over one collection of objects, or over many collections, allowing the inspection of relationships between objects in these collections. The Java Query Language (JQL) provides first-class object querying for Java, and we present the full details of JQL in chapter 3. A simple object query in JQL looks like this:

```
selectAll(String s : someStrings | s.length() > 5);
```

This query performs a filter operation — it selects all `String` objects, which are contained in the collection `someStrings`, and which have a length greater than 5. It returns a `List` of `String` objects. Queries over multiple collections look and behave much the same, and return tuples of objects. We describe the syntax of JQL queries in more detail in Section 3.1.

Queries are more succinct and readable than their equivalent loop implementation. They are also more abstract, and this offers the capability for dynamic reconfiguration of the query expression. They also allow for the possibility of more efficient evaluation of *joins* between collections, with less programmer effort than manual implementations. A join between collections is an operation that compares or operates on combinations of objects from each collection. For an example of such a join, consider Figure 2.5, which shows a simple class diagram of Developers, Employees, and Projects. Many developers work on a project; many employees also work on a project. Developers are not necessarily employees, however — many contributions are made from volunteers and external companies. Not all employees working on a project are developers, either, as some are in marketing, management, etc.

Developer-employees are therefore represented in this decomposition with two objects, which are semantically related but entirely independent within the system. A problem, then, is finding those developers who are




```
class Dev{String email; Project currentProject; ...}
class Employee{String email; int salary; ...}

class Project{
  String name;
  HashSet<Bugs> bugList;
  ...
}
```

Figure 2.5: A UML diagram describing developers that work on a project, and employees that also work on a project, as well as a simplified implementation.

also employees. A new collection could be created to hold these developer/employee pairs, but this collection would need to be maintained with additions and removals from the developer and employee sets, increasing the complexity of other code that deals with these sets. Another possibility could be the use of multiple inheritance or interfaces, using a class that inherits or implements both developer and employee. This is feasible, but this solution complicates further the addition and removal of objects from either set — removing a developer-employee object from the developer set means its corresponding entry in the employee set needs to be converted to a normal employee instance, and vice versa. A straightforward solution, without using and maintaining another collection or using multiple inheritance, is to construct the set of developer-employee pairs when it is needed. The following Java code is one approach:

```
List<Object[]> matches = new ArrayList<Object[]>();
for(Dev d : devs){
  for(Employee e : emps){
    if(d.email.equals(e.email)){
      matches.add(new Object[]{d,e});
}}}
```

This is a join of the collections of developers and employees, on the email field. The following object query is a more succinct version of this code:

```
List<Object[]> matches;
matches = selectAll(Dev d:devs, Employee e:emps |
                            d.email.equals(e.email));
```

The results of this query will be exactly the same as the results of the manual loop code. However, the use of a query for this version allows optimization opportunities that are not feasible using a manual implementation. These joins correspond with our *Product* category of loop operations, and whilst these loop operations are the least common found in our study, they can easily be the most expensive.

2.3 Efficient Joins

In addition to being more readable, the query approach is more abstract. That is, whilst the loop version specifies exactly how this join is to be evaluated (via nested loop), using a query the exact detail of how to evaluate the join is abstracted away. This saves the developer from both having to think about how to do the join and from having to implement it. Instead, the query evaluator can select a method for joining the two collections together, making use of advanced optimizations the developer may consider too complex or time-consuming to bother with.

One such optimization is the *hash join*. The hash join is useful in situations where the number of matches for a query are likely to be small. In the case of finding developer-employees, the 'email' field is unique to a particular person, so there should only be one developer with the same email as an employee. To avoid evaluating all the combinations of developers and employees (i.e., the *cross product* of the sets), we can exploit the fast lookup of hash tables. A hash-map is built from the smallest of the two collections, which maps the value of the joining field (`email`) back to the object it is from. Then, the other collection is iterated over, looking up each object's email value in the hash-map, and adding the returned object to the result tuples (if any object is returned).

Figure 2.6 illustrates a hash-join implementation of the original loop code. This code offers better performance than the nested loop implementation — we generally expect $O(n+m)$ rather than $O(nm)$, where n and m are the size of the `devs` and `emps` collections. It is, however, much longer than the nested loop implementation, harder to understand, and harder to debug. We consider it quite unlikely that developers would generally take the time to manually implement this kind of join.

JQL's query evaluator supports join techniques such as the hash join and the sort join; these are outlined in more detail in Section 3.4. These joins are applied automatically by the query evaluator whenever possible.

```
//  selectAll(Dev d : devs, Employee e : emps
//                    | d.email == e.email);
HashMap<String, List<Employee>> tmp = new HashMap<..>();
for(Employee e : emps){
  List<Employee> es = tmp.get(e.email);
  if(es == null){
    es = new ArrayList<Employee>();
    tmp.put(e.email,es);
  }
  es.add(e);
}

List<Object[]> matches = new ArrayList<..>();
for(Dev d : devs){
  List<Employee> es = tmp.get(d.email);
  if(es != null) {
    for(Employee e : es){
      matches.add(new Object[]{d,e});
    }
  }
}
```

Figure 2.6: A hash join implementation of a simple query. The code first builds a map out of one of the input collections, and then iterates the other collection looking up matches.

2.4 Join Ordering

Another detail that querying abstracts away is the ordering of conjunctions in a query predicate. For example, this query finds all developers who are employees, who are currently working on an active project:

```
List<Object[]> matches;
matches = selectAll(Dev d:devs, Employee e:emps,
                    Project p: activeProjects |
                         d.email == e.email &&
                         d.currentProject == p);
```

There are two parts to this query's predicate: `d.email == e.email` and `d.currentProject == p`. We view these parts as *stages* in a *pipeline*. The objects which pass one stage go on to the next to be evaluated. Evaluating the most *selective* stage first can greatly increase the efficiency of this query. Selective stages are those which return few results; fewer results from one stage mean fewer inputs to the next, and fewer evaluations to perform in total. This is explored in more depth in Section 3.2.

There are two possible orderings for the evaluation of this query. Which ordering is optimal depends both on the relative sizes of these sets, and the selectivity of each stage. Unfortunately, neither ordering is optimal in all cases. Consider a situation where every developer has a different project in the `activeProject` collection, but where very few developers are also employees. In this case, it makes sense to do the `d.email == e.email` stage of the query first, as this eliminates many developer objects from consideration for the second stage. On the other hand, were there very few projects and many of the developers were also employees, the other ordering would be better. Knowing the selectivity of each stage is critical for deciding the order of evaluation intelligently.

Even assuming a programmer implementing this query has the domain knowledge to accurately estimate these selectivities at deployment time, the factors which determine them (and thus, the optimal ordering)

are not static. Projects begin and end, employees come and go, etc. Attempting to account for these external factors while coding is practically impossible. The impact of these factors can, however, be detected at runtime. One technique for this is a sampling heuristic: by testing the query for small numbers of objects, we can get an idea of the composition of the collections we are querying, and make decisions about how to order the query evaluation based on the sample. Such a heuristic is extremely unlikely to be used in a manual implementation, however, as it entails extra code. The best that can be reasonably hoped for, then, with a static ordering is being right most (but not all) of the time.

JQL's query evaluator uses heuristics to determine the evaluation order for the stages of its queries. The query evaluator can weigh these heuristics at evaluation time, including a sampling heuristic to make useful estimations of the optimal evaluation order (outlined further in section 3.5). This frees developers to simply specify what they want to find, without having to worry about the fiddly details of performance optimization, and instead concentrate on interesting problems.

2.5 Caching and Incrementalization

Queries are likely to be used more than once. After a query has been evaluated, we can *cache* the results of the query — simply storing a copy of the result set before it is returned to the program, and returning this stored result set instead of re-evaluating the query when it is called again. Any loop operation with a filtering component which can be expressed as a query could stand to benefit from having these results cached.

Once results are cached, they can be kept up to date with a technique called *incrementalization*. Incrementalization maintains the cached results in line with changes to objects within the program. For example, if we are querying the `emps` set of `Employee` objects, and a new `Employee` is added to the set, the incrementalizer checks the new object against the

query expression and adds the new object to the cached results if the expression returns true.

JQL's caching and incrementalization facilities are described in more detail in Chapter 4.

2.6 From Loops to Queries

Having briefly introduced JQL, its capability for dynamic reconfiguration, and the added benefit of caching queries, we now describe how object querying as provided by JQL correlates with the loop categories described in Section 2.1.

Filter

Queries correlate very naturally with simple filters. Consider the filter example from Figure 2.1 — this can easily be rendered as a JQL query like so:

```
doAll(Robot r : robots | !r.isDead()){r.method();}
```

Note, `doAll` is syntactic sugar for a "`for(..: selectAll(..))`" loop. When implemented using queries, filter operations such as this can benefit from query caching and incrementalization. Normally, a developer might manually create a collection of 'live' robots to speed up these operations; this saves looping over all the robots. However, this introduces complications for the robot class — we now need to ensure when we set a robot to 'dead', that we also remove it from the set of 'live' robots. This causes inconvenient coupling between the collection and its members. If a query is used for this operation, however, this collection of 'live' robots is created automatically by the caching system. Furthermore, the maintenance of this set is performed automatically by the incrementalization system, relieving the added complexity a manual implementation incurs.

More complex filters can also map to queries. Our earlier example using a switch statement, from Figure 2.2, maps to multiple JQL queries, one for each case:

```
doAll(String s : names | s.charAt(0)=='D') {
  item = new TreeItem();
  item.setText(s.substring(1));
  item.setImage(new Image("images/folder.gif"));
}
...
doAll(String s : names | s.charAt(0)=='F') {
  item = new TreeItem();
  item.setText(s.substring(1));
  item.setImage(new Image("images/file.gif"));
}...
```

These filters can also be cached and incrementally maintained. There are two caveats for this translation, however. Firstly, the order in which objects in the collection have their operation applied may differ from the original loop (and so, additional measures may need to be taken to achieve equivalent semantics); and, secondly, as seen in this case, the queries can incur code duplication (although this could easily be factored out into a method call).

Product

Product operations also map to queries, and accrue additional benefits. The product operation given in Figure 2.3 can be expressed as the following JQL query:

```
doAll(Robot r:robots, Robot dead:deadRobots |
      !r.isDead() &&
     (r.team == null || r.team != dead.team)) {
```

```
    r.scoreSurvival();
  }
```

This query version can take advantage of query ordering and efficient join strategies, as well as caching and incrementalization, to optimize the evaluation of this operation. Product operations can be easily the most time-consuming of loop operations, so even though they are relatively rare (as shown in Table 2.2), their optimization could still make a large difference to overall execution time.

Map

Map operations are performed in JQL using Java's enhanced `for` loop, as it already provides a suitable abstraction for map style operations.

Reduce

Reduce operations are not expressible as JQL queries. This is simply because these operations do not select elements from a set; additional functionality could be added to JQL queries to support these operations, but this is an orthogonal concern from the other types of queries.

Other

Operations we have categorized as 'Other' are also not readily expressible as queries, as the operation they perform either depends on the ordering of the items within the collection, or operates on more than one item from the collection at a time.

Having shown, then, that a fair proportion of operations over collections are readily expressible as queries using JQL, we will now describe JQL in more depth.

Chapter 3

JQL

The Java Query Language provides straightforward object querying for Java. We have designed JQL as an extension to Java, and implemented a prototype JQL system, in order to investigate the performance and usability characteristics of object querying. This chapter describes the syntax of the Java Query Language, some performance optimizations it provides, and how our JQL prototype operates. It also presents performance data comparing a querying approach against some 'hand rolled' implementations for solving certain problems. This is to show the variations in performance that implementations can offer, and how our JQL prototype performs.

3.1 JQL Syntax

JQL's syntax is influenced by the 'enhanced' for loop introduced in Java 1.5. Here is a simple query:

```
List<Object[]> matches;
matches = selectAll(String s:words, Integer i:lengths |
                              s.length() == i);
```

This query finds all strings from a collection `words`, that have a length that is contained in `lengths`. JQL's syntax is also influenced by set and list comprehension constructs from other languages (e.g. Python, Haskell). A grammar for queries is given in Figure 3.1. Queries have two parts — the domain variable declaration, and the query expression. We separate these parts with a vertical bar ('|'). The domain variable declaration states which collections we are querying over, and associates a 'domain variable' with each collection. Our example query has a domain variable `s`, ranging over `String`s from the collection `words`, and a domain variable `i` ranging over `Integer`s from the `lengths` collection. The query expression, after the bar, is a Java expression in conjunctive normal form, that uses these domain variables, and evaluates to either true or false. Query expressions can use any values available in the current scope, and methods may be called as normal. Queries can be declared with two keywords in JQL: `selectAll` and `doAll`. `selectAll` queries return the results of the query as a collection of objects. `doAll` queries are `selectAll` queries with an attached block of code. This block of code will be applied to every result tuple in the result set of the query, and the domain variables used in the query expression can also be used in this block.

3.2 JQL Semantics

Evaluating a JQL query tests the query expression over the cross-product of the objects from the source collections given. It then returns a list of tuples (i.e., arrays) of objects, and these tuples consist of all possible combinations of objects from the source collections where the query expression holds. In the case of a query with only one source collection, a flat list of objects is returned.

Although the returned tuples are guaranteed to be the complete set of combinations of objects for which the query expression holds, it is not guaranteed that the query expression will be *actually* evaluated over any

F ::= forall($d \mid Q$);
D ::= doall($d \mid Q$)$\{SS\}$
d ::= $C\ l : c$; [d;]
Q ::= e
| e && Q
| e || Q
SS ::= $S\ [SS]$
ev ::= $v \mid l$
e ::= (e_1 ($> \mid < \mid >= \mid <= \mid == \mid !=$) e_2)
| ev
| $ev.f$
| $ev.m$
| n

v	$\in$	Var	local variables
f	$\in$	$Field$	fields
m	$\in$	$Methods$	methods
n	$\in$	$Consts$	constants
C	$\in$	$Classes$	class names
c	$\in$	$Collections$	collections
l	$\in$	$DVars$	domain variable names
S	$\in$	Statements	java statements

Figure 3.1: An abridged grammar for these first-class query constructs that extends the Java grammar given in [16].

particular combination of objects. Optimizations such as short circuiting allow the query optimizer to skip many redundant evaluations. On the other hand, the use of a sampling heuristic (as outlined in Section 3.5) to determine optimal join orderings could cause some combinations of objects to be evaluated more than once, or evaluated when they would otherwise be skipped. This unpredictability has ramifications when using method calls with side-effects in a query expression, as it cannot be predicted which objects will have methods called on them, and how many times. We therefore assume that methods in a query expression are pure (i.e., side-effect free).

Unlike for a regular Java expression, boolean operators do not imply an ordering for the execution of their operands. The ordering of these operations can greatly affect the performance of queries with more than one source collection, and the factors which decide the best ordering (such as the size of input sets) may not be known until run-time. The ordering decision is therefore delayed until run-time, when the query evaluator can examine these factors. This is detailed in Subsection 3.5.

3.3 JQL Implementation

Our implementation of JQL has two main components. The first is a front-end, which compiles JQL expressions into Java code. The second component is the query evaluation back-end, which is responsible for evaluating the query and choosing the most optimal evaluation strategy it can.

JQL Compiler

The JQL compiler is relatively straightforward — it compiles JQL expressions into equivalent Java code, which can then be compiled by a normal Java compiler. This Java code binds domain variables to collections, and builds a tree representation of the query expression, which the evaluator

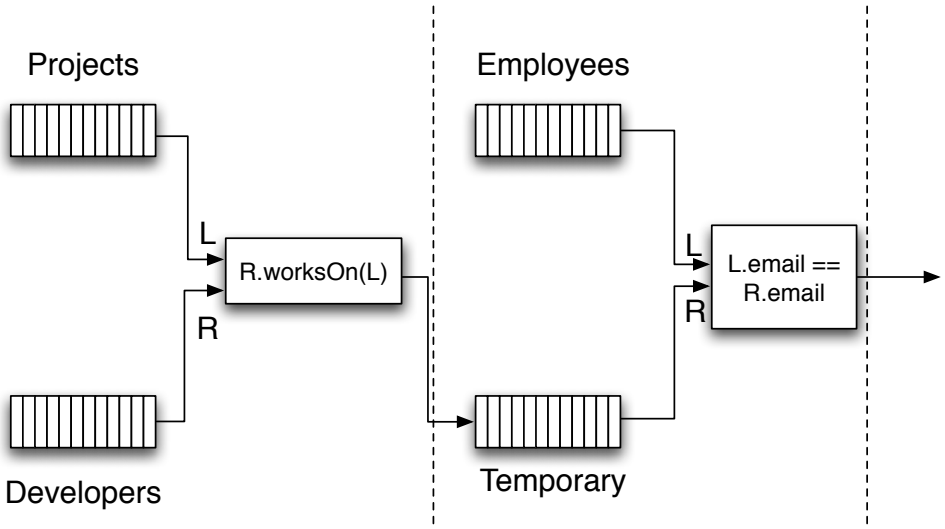


Figure 3.2: Illustrating a query pipeline.

uses at runtime. The compiler 'inlines' queries which only use one domain variable; they are compiled into normal loops, but with hooks to allow for query caching. Inlined queries cannot have their stages dynamically ordered, but dynamic ordering is of almost no importance to single-variable queries (whilst different orderings may 'short-circuit' evaluation for individual objects more quickly, the gain is minute compared to the performance gain of inlining the query). Using the compiler also allows for checking of the well-formedness of the query — for example, that query expressions only use domain variables that are declared for that query.

Query Evaluator

The core of query evaluation is carried out through the query's 'Query Pipeline', as shown in Figure 3.2. A query pipeline consists of a series of stages, each corresponding to a part of the query expression. Each stage can take either one or two input sets of tuples, and for all stages except the very first, one of these input sets is the result set of the previous stage

in the pipeline. This structure is known, in the database community, as a 'left-branching tree', and we impose it to simplify the problem of ordering stages. In each stage, if there is more than one input set, the input tuples are combined (we describe the join techniques used for this combination shortly), and the stage's expression is evaluated using the values in the combined tuple in the place of their corresponding domain variables. If the expression evaluates to true, the combined tuple is added to the stage's result set. The set of tuples which pass the final stage are returned to the program as the results of the query.

3.4 Join Types

When a pipeline stage has more than one set of input tuples, the JQL query evaluator has to decide how to combine the input sets. JQL makes use of three main join types: Nested Loop Join, Hash Join and Sort Join. These join types have different performance characteristics, and are applicable for different expressions.

Nested Loop Join

The nested loop join is the simplest join type that JQL uses, but it is also the slowest. It is the fallback join used when neither sort join nor hash join can be used. It combines the tuple lists by means of a nested loop over each source. This then requires |**L**| x |**R**| query evaluations, where **L** and **R** are the input sets. In JQL, joins which find pairs of objects which are not equal to each other are implemented using nested loops. Other types of joins can use a nested loop, however it is usually preferable to use another join type.

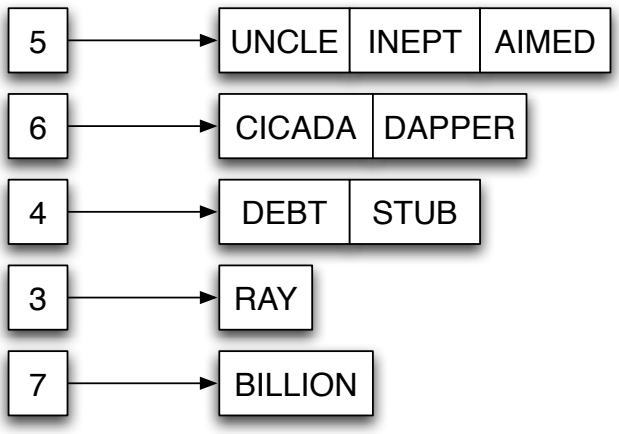


Figure 3.3: A map from join value to objects, as would be constructed by a hash join. In this case, we map `String.length()` back to `String` objects.

Hash Join

Hash joins are usually the fastest join type. They can only be used for equality expressions (== and `equals()`), however, as they do not directly evaluate their join expression per se. Instead, hash joins exploit the property that two Java objects which are equal must have the same `hashcode`. This means we can store some value in a hash-map, using an object as the key, and if we get that value back using another object as the key, the two key objects are equal. The join is performed by first building a hash-map from one of the input sets. This maps from the value being joined upon to the objects that hold that value. Once this map has been built, we then iterate over the other set, looking up the value being joined upon for each object from this set. Any objects returned from the hash-map using this value as the key pass the equality join, and can be added to the result tuples. In the example query from section 3.1, the hash-map would map from the values of the `length()` method call to the set of `String` ob-

jects that have that length. Figure 3.3 shows the structure of the resulting map. Then, we iterate over the set of `Integer`s, looking each one up in the map, which returns the set of `String` objects that return that value when `length()` is called on them. In the worst case, this can take $O(nm)$ time, but in practice it is most likely to be linear in the size of the result set.

Sort Join

The Sort Join, while usually not as fast at the Hash Join, is a large improvement on the Nested Loop Join. Sort Joins are used for comparison joins: greater than, less than, greater than or equal to, less than or equal to, and, finally, for calls to `compare()`. Sort join proceeds much like a Nested Loop join, except it first sorts the input tuples (whether they are sorted in ascending or descending order depends on the join's expression) on the value being joined. Then, for each tuple in **L**, it scans **R** for the first tuple where the expression holds. Once this is found, the rest of tuples from **L** can be combined with that tuple from **R** and added, without doing the comparison. We then check the next tuple in **R**. If the match fails, we move to the next tuple in **L** and try again. Figure 3.4 gives an overview of the sort join code for join two ascending lists of integers. This requires $O(|\mathrm{L}|)$ evaluations of the expression, and each sort is $O(n \log n)$. In the worst case, however, the sort join can still take $O(|\mathbf{L}||\mathbf{R}|)$ time, when adding the matching tuples to the result-set is taken into account. JQL's sort join clones the collections it joins before it sorts them, and so does not modify the ordering of the collection objects which are passed to it.

3.5 Ordering Strategies

The time taken to evaluate a query can be dramatically altered by the order in which the stages in the query pipeline are processed. The cost of each stage in the query pipeline is determined by the size of its inputs, and

```
//sort both lists in ascending order
Collections.sort(leftSide);
Collections.sort(rightSide);

ArrayList<Integer> results = ....;
int leftpos = 0;
Iterator rit = rightSide.iterator();
Integer r = rit.next();
Integer l = leftSide.get(leftpos);
while(true){
  if(l > r){
    results.add(new Integer[]{l,r});
    // l > r for this l, so add the rest
    for(int i = leftpos+1; i < leftSide.size(); i++){
      matches.add(new Integer[]{leftSide.get(i),r});
    }
    if(rit.hasNext()){
      r = rit.Next();
    }else{
      break;
    }
  }else{
    leftpos++;
    if(leftpos > leftSize.size()){
      break;
    }else{
      l = leftSide.get(leftpos);
    }
  }
}
```

Figure 3.4: A simplified view of sort join code, that joins two ascending lists of integers.

by the type of its join expression. As each stage, except the first, has a previous stage as one of its input sets, these stages' costs depend on the size of the result sets from the preceding stages. The *s*electivity of each stage is important; we use the term to mean the ratio of tuples which do not match to the input size. Highly selective stages pass fewer results on to the next stage, which then has smaller input sets than if its preceding stage was less selective. This can have an effect all the way down the pipeline. Unfortunately, finding the optimal ordering for a pipeline of n stages requires searching $n!$ possible orderings (indeed, this is known to be an NP-complete problem). Worse, it is necessary to know the selectivity of each stage before searching for the optimal order, and this cannot be known for sure without evaluating the query, as it depends on the specifics of the objects provided as input.

JQL uses two heuristics to estimate the selectivity of a query. The first is a fixed heuristic, for each possible join operation. The heuristic uses the following selectivity values: 0.95 for `==` and `equals()`; 0.5 for $<$,$\leq$,$>$,$\geq$ and `compare()`; 0.2 for `!=`; and 0.1 for arbitrary methods. The second approach is a sampling heuristic, similar to that used by Lencevicius et al. [26], which evaluates a small number of randomly selected tuples from the inputs and uses the number that pass as an estimate of the selectivity of that join. Even with very small sample sizes, we find that useful results can be obtained.

Once it has estimates for each join in the query pipeline, the JQL query evaluator can then attempt to apply an ordering heuristic. At present we have two: an **e**xhaustive search heuristic and a **m**aximum selectivity heuristic.

Exhaustive Search Heuristic

The exhaustive ordering heuristic searches all $n!$ possible orderings of a query pipeline, adding together the cost for each join, and estimating the likely output size of each join for use in the cost calculations of the next

join. The exhaustive ordering heuristic is usually the more reliable ordering strategy, but it is expensive. Furthermore, it is not infallible; its results depend on the accuracy of the estimated selectivity for each stage, and on the estimates of the costs of the join types.

Maximum Selectivity Heuristic

The maximum selectivity heuristic simply orders the stages in the pipeline based entirely on the estimated selectivity of each stage, with the most selective stages going first. This avoids the exponential search of the exhaustive heuristic, however it often produces less optimal orderings. Just like the exhaustive heuristic, the accuracy of this heuristic is also dependent on the estimated selectivity for each stage.

3.6 Performance

Whilst querying provides useful opportunities to abstract away details and reduce developer effort, it is still important that the performance of querying remain at least competitive with that offered by conventional, manual implementations. For each loop there is one or more optimum implementations; ideally, JQL could match this performance. Unfortunately, pipeline setup and searching for good query evaluation orders adds overhead. However, given how difficult these optimum loops can be to write, and how unlikely it is the average programmer will do so, we consider that performance that is even competitive with an optimum implementation is an indication of success. In this section we investigate how JQL's performance compares to hand-coded implementations of three queries of varying complexity.

In all experiments which follow, the experimental machine was an Intel Pentium IV 3.2GHz, with 1.5GB RAM running NetBSD v3.99.11. In each case, Sun's Java 1.5.0 (J2SE 5.0) Runtime Environment and Aspect/J

version 1.5M3 were used. `System.currentTimeMillis()` was used to perform timing. The source code for the JQL system and the three query benchmarks used below can be obtained from `http://www.mcs.vuw.ac.nz/~darren/JQL/`.

3.6.1 Study 1 — Query Evaluation

The purpose of this study is to compare JQL's query evaluator with the performance offered by equivalent manual implementations. We used three queries of different complexities as benchmarks. These queries are detailed in Table 3.1. Three manual implementations of these queries were written: HANDOPT, HANDHASH and HANDPOOR. HANDOPT represents the best implementation we could write. It uses hash and sort joins where appropriate, and has the optimal join ordering (for these particular input sets) hardcoded in without having to search. HANDHASH uses a hash join, but a sub-optimal join order. HANDHASH is quite a realistic implementation for an advanced developer who is aware of the value of hash joins, but has failed to notice the optimal ordering (especially likely as the optimal ordering for the `ThreeStage` query is unintuitive). HANDHASH was not used for `OneStage`, as there are no ordering concerns and thus it is the same as HANDOPT. HANDPOOR is a straight-forward nested loop implementation using the worst possible join ordering. HANDPOOR is pessimistic, but certainly a possible outcome for programmers who are in a hurry or distracted.

Our investigation aims to find the spread of performance for various different hand-coded implementations, and where JQL's automatic implementation fits in. Given the difficulty and time investment involved in both implementing the specialized hash and sort joins, and the additional complexity (if not impossibility, recall Section 2.4) of selecting a perfect join ordering strategy, comparing JQL against the whole range of possible manual performances is interesting.

Name	Details
OneStage	`selectAll(Integer a:L1, Integer b:L2 | a == b);` This benchmark requires a single pipeline stage. Hence, there is only one possible join ordering. The query can be optimized by using a hash-join rather than a nested loop implementation.
TwoStage	`selectAll(Integer a:L1, Integer b:L2, Integer c:L3 | a == b && b != c);` This benchmark requires two pipeline stages. The best join ordering has the joins ordered as above (i.e. == being first). The query can be further optimized by using a hash-join rather than a nested loop implementation for the == join.
ThreeStage	`selectAll(Integer a:L1, Integer b:L2, Integer c:L3, Integer d:L4 | a == b && b != c && c < d);` This benchmark requires three pipeline stages. The best join ordering has the joins ordered as above (i.e. == being first). The query is interesting as it makes sense to evaluate `b != c` before `c < d`, even though the former has lower selectivity. This query can be optimized using a hash-join as before, and a sort-join for the `c < d` join.

Table 3.1: Details of the three benchmark queries

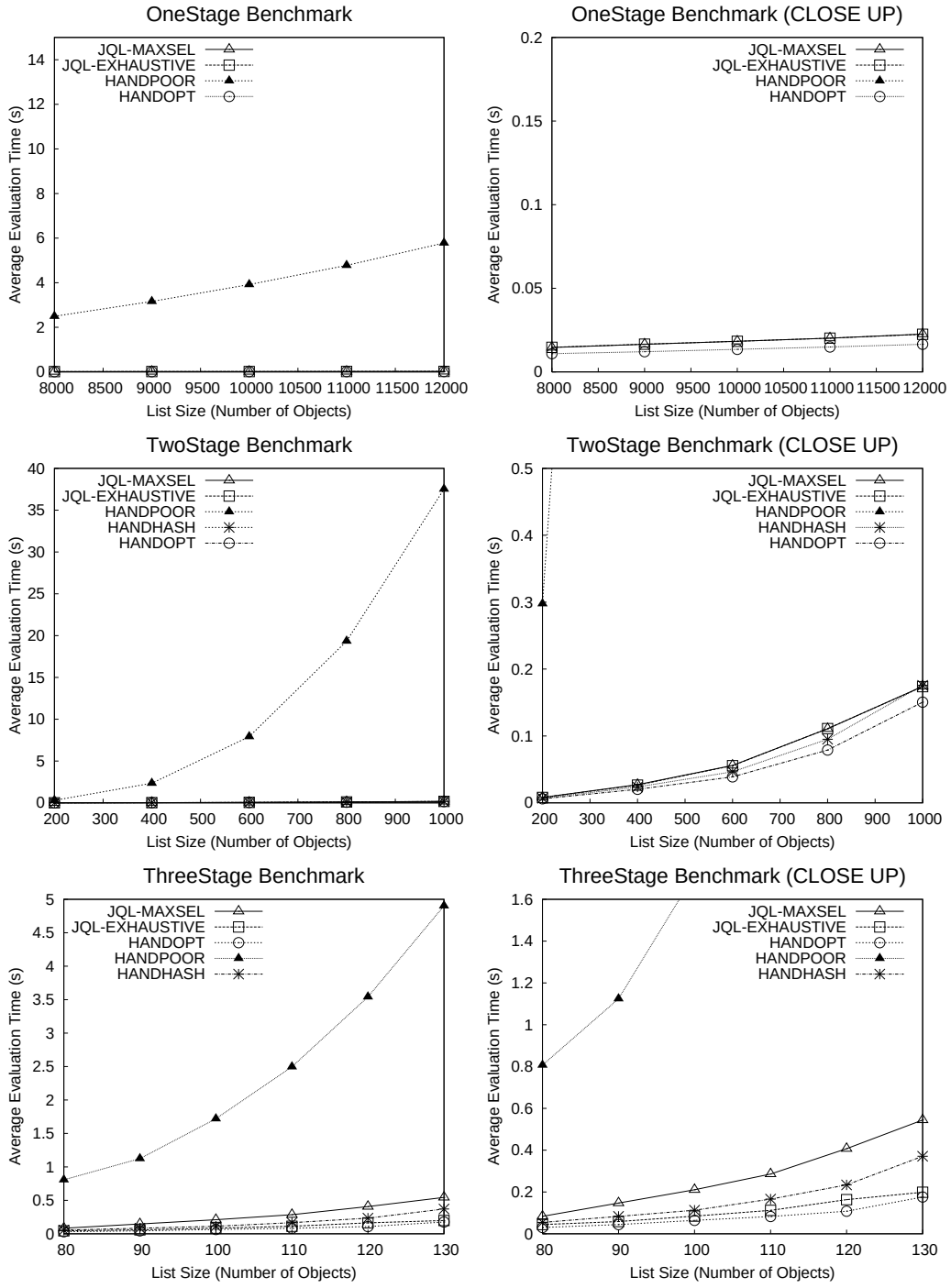


Figure 3.5: Experimental results comparing the performance of JQL with different join ordering strategies against the hand-coded implementations. Data for the OneStage, TwoStage and ThreeStage benchmarks are shown at the Top, Middle and Bottom respectively. The charts on the right give close ups of the three fastest implementations for each benchmark. Different object sizes are plotted to show the general trends.

Experimental setup.

The queries range over the lists of `Integer`s `L1`, ..., `L4` which, for simplicity, were always kept the same size. Let n be the size of each list. Then, each was generated by initializing with each integer from $\{1, \ldots, n\}$ and randomly shuffling. Then, the worst case time needed to evaluate `OneStage`, `TwoStage` and `ThreeStage` queries is $O(n^2)$, $O(n^3)$ and $O(n^4)$, respectively.

For each query benchmark, four implementations were tested: the three hand-coded implementations (HANDOPT, HANDPOOR, HANDHASH); and, the JQL query evaluator using the MAX_SELECTIVITY and EXHAUSTIVE join ordering strategies. For all JQL tests, join selectivity was estimated using the fixed heuristic outlined in Section 3.5, but not the sampling approach. The reason for this is simply that, for these queries, the two approaches to estimating selectivity produced very similar results.

Each experiment consisted of measuring the average time taken for an implementation to evaluate one of the query benchmarks for a given list size. The average time was measured over 10 runs, with 5 ramp-up runs being performed beforehand. These parameters were sufficient to generate data with a variation coefficient (i.e. standard deviation over mean) of ≤ 0.15 — indicating low variance between runs. Experiments were performed for each query and implementation at different list sizes (i.e. n) to gain insight into how performance varied with n.

Discussion

The results of the experiments are shown in Figure 3.5. The main point of interest is the large performance spread between an optimized implementation and a poor one, even for a simple query. While the performance of JQL is always slower than HANDOPT (a given, as HANDOPT has its optimal join strategy hard coded), it is always far better than HANDPOOR. When the exhaustive strategy is used for the `ThreeStage` query, it per-

forms better than HANDHASH and comes very close to HANDOPT. We argue, then, that JQL's good performance regardless of programmer ability is very attractive. Furthermore, the conciseness of the query implementation compared to the optimal implementations is important — Appendix B contains the source code for the optimal implementation of the `ThreeStage` benchmark, which is quite long and difficult to immediately grasp. The corresponding query, on the other hand, is only a few lines long.

The `ThreeStage` benchmark is the most complex of those studied and highlights a difference in performance between the MAX_SELECTIVITY and EXHAUSTIVE join ordering strategies used by JQL. This difference arises because the MAX_SELECTIVITY heuristic does not obtain an optimal join ordering for this benchmark, while the EXHAUSTIVE strategy does. In general, the EXHAUSTIVE strategy is the more attractive. However, it is important to remember that it uses an exponential search algorithm and, hence, for queries with a large amount of joins this will certainly require a prohibitive amount of time. In these cases the maximum selectivity heuristic would be more appropriate.

3.6.2 Study 2 — Dynamic Join Ordering

Study 1 has established that an extremely important aspect of querying performance is the order in which joins are evaluated. The purpose of this study is to investigate the ability of JQL's *selectivity sampling* capability to adapt to different situations; i.e. those where the optimal join ordering for a query changes between query evaluations. This occurs due to changes in the underlying data in the source collections for the query.

Experimental Setup

For this study, we used a query specially set up to allow us to 'tweak' the selectivity of its joins. A two-stage query was used, similar to the

`TwoStage` benchmark from the previous section. The exact query expression used in this case was

```
selectAll(Integer a:L1, Integer b:L2, Integer c:L3 |
                  a == b && b == c);
```

and the contents of the lists `L1`, `L2`, and `L3` were selected to enforce certain selectivities on the joins. Each list was filled with 100 values. Initially, `L1` and `L2` were filled with the same value x, while `L3` was filled with value y. This setup meant that the first `a == b` join had low selectivity, as all `a`s and `b`s were equal to x and to each other (and so all possible pairs of `[a,b]` were passed onto the next stage). As none of them were equal to y, the second join had a very high selectivity, and no tuples made it through this stage at all. We then altered the selectivity of these two joins by changing a proportion of `L2`'s members to y. This simultaneously increased the selectivity of the first join and lowered that of the second.

We compared JQL's exhaustive-sampling strategy, which samples the selectivity of joins before exhaustively enumerating all possible orderings, with JQL's performance when each of the two possible orderings are hard-coded into the program. Hash joins were disabled for each query, to remove the effects of that optimization from consideration.

The lists used as input were all of size 100, and proportion of values replaced with y was increased by 0.05, ranging from 0 to 1.0. The average time was measured over 15 runs, with 5 ramp-up runs being performed beforehand.

Discussion

Results for this experiment are shown in Figure 3.6. The measurements labeled **1-2** are for the query evaluated with `a==b` first; **2-1** is the other ordering. The main observation is that the performance of JQL using sampling follows the shape of the best results from both fixed orderings, rising at 0.5 then falling. Whilst in most cases the overhead for the sampling is

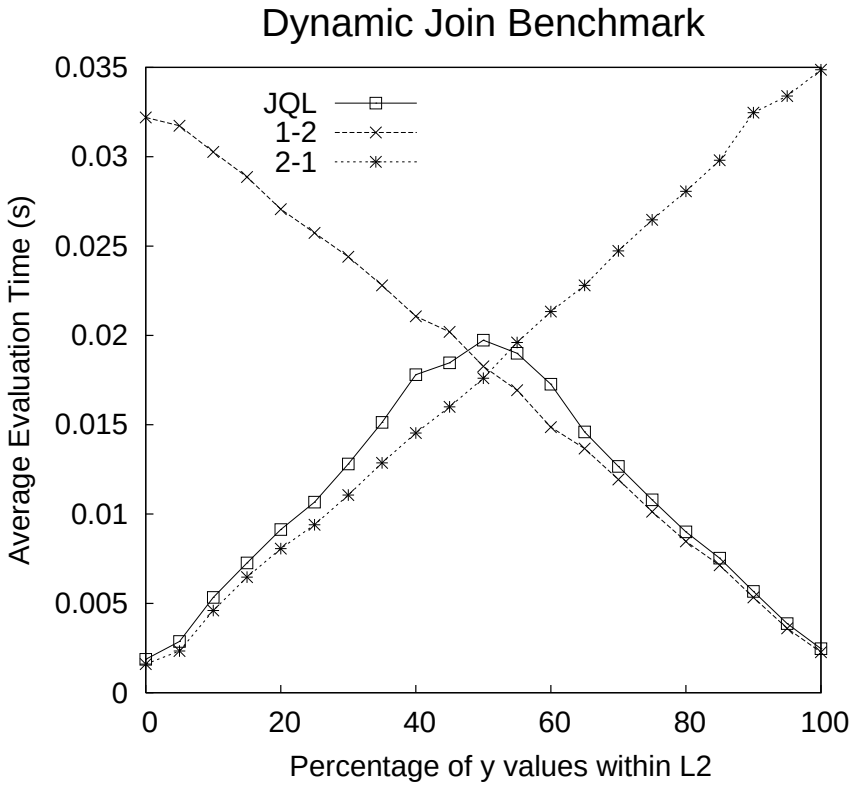


Figure 3.6: Experimental results comparing JQL using three different join ordering strategies for evaluating a two-stage query. Two static and one dynamic strategy are used.

minimal, the sampling implementation's line 'bulges' between the 40% and 60% proportions. Inspection of the data reveals that this is due to cases where the join configuration fails to find the correct ordering. Finding the correct configuration becomes less likely the closer the ratio is to 0.5 (however, as the ratio approaches 0.5, the performance difference between orderings also decreases. At exactly 0.5 the configuration no longer matters). For this experiment, the query evaluator's worst performance was at a ratio of 0.55, where it found the correct ordering for ten out of fifteen runs.

Chapter 4

Caching and Incrementalization

Chapter 3 has demonstrated JQL's usefulness when joining multiple collections together. Adding caching means that query results can be obtained much faster, while incrementalization means that these cached results are updated in line with changes in the program. In this chapter we also outline the implementation of the caching system used with JQL. We give an overview of how the incrementalization works, and present investigations into the usefulness of using JQL to implement a flexible data type and the impact of incrementalization overhead.

4.1 Caching and Incrementalization

Caching is a well understood concept; we save the result of an expensive computation (in this case, a query evaluation), and whenever that computation is to be performed again, we simply return the saved result. It is a direct trade-off of space for time. An *incrementalized* cache is slightly more complicated. When using a cache, it is important that the results returned from the cache are identical to those that would be returned by the 'real' operation. If the inputs to the operation can change, and in doing so affect the result of the computation, then it becomes necessary to update the cached result.

As an example, take the query from Section 2.2, that was used to match `Developer`s with `Employee`s:

```
List<Object[]> matches;
matches = selectAll(Dev d : devs, Employee e : emps |
                                d.email.equals(e.email));
```

We can cache the result of this query by simply storing the set of matches. Once this is done, however, we need to observe any changes to the sets `devs` and `emps`; a change to either of these sets could potentially invalidate the results we have stored. If a new `Developer` object is added which corresponds to an existing `Employee`, then there is a new match for this query, which must be reflected in the stored results. One way to do this would be to re-run the entire query and store the results again. This maintains the cache's validity, but is expensive. An alternative is to simply determine the change to the input and combine this with the previously cached result to derive the new result. In this case, the change is the addition of a `Developer`. We combine this change with the cached result by checking it against the `emps` set for a matching employee, and adding a pair if we find them.

As updates are made to the cache in small increments, this is termed incremental updating. A cache which is incrementally updated is termed an *incrementalized cache*. Incrementalized query caching is a powerful construct, and we now outline the use of object querying, augmented with incrementalized caching, for implementing data types with flexible interfaces.

4.1.1 Flexible Data Types

Abstract Data Types (ADTs) support *implementation hiding*. They offer an interface to the data they contain, without exposing the details of how they are implemented. Common examples include Lists, Stacks, Queues and Maps. Object querying provides a high level of abstraction that can

```
interface Tree{
  public Tree(int numchildren);
  public Object[] getChildren(Object x);
  public boolean addNode(Object x);
  public boolean addChild(Object x, Object y);
}
```

Figure 4.1: A simplified tree interface.

subsume many common ADTs, abstracting away the choice of which ADT to use. One view, put by Liu et al in [32], is that all operations on data can be seen as either *queries* or *updates*. Queries return information about the data within a type (such as a `contains()` call on a collection), whereas updates change the data (such as `add()` calls on a collection). In this view, then, the interface to an ADT is a predetermined set of queries, and the implementation of the ADT is often optimized entirely for this set of queries.

Consider a tree ADT making use of the interface in Figure 4.1. A simple implementation of this interface could be a `Set` of tuples of size $n + 1$, where each tuple contains an object at position 0, and the object's children at positions $1..n$. A simplified implementation of this is given in Figure 4.2. For an object `A`, with children `B`,`C` and `D`, our tree has a tuple `[A,B,C,D]`. If we add children `E` and `F` to `B`, and some children to `C` as well, our `Set` becomes:

```
List<Object[]> tree = {[A,B,C,D],[B,E,F,null],
                       [C,G,H,I]}
```

We can inspect and manipulate this data structure by simply iterating over the set of tuples. Trees are commonly manipulated by 'walking' them, i.e., starting at the root node and following its children over the tree. Finding the children of a node requires iterating through the `Set` of tuples, checking position 0 in the tuple until you find the target node. This is an $O(n)$

```
class SetImpl implements Tree{
  Set<Object[]> nodes;
  int numchildren = 4;
  public Object[] getChildren(Object x){
    for(Object[] n : nodes)
      if(x.equals(n[0])){
        Object[] children = new Object[numchildren];
        System.arraycopy(n,1,children,numchildren);
        return children;
      }
  }
  public boolean addNode(Object x){
    for(Object[] y : nodes)
      if(y[0] == x){return false;}
    Object[] n = new Object[numchildren+1];
    n[0] = x; nodes.add(n);
    return true;
  }
  public boolean addChild(Object x, Object y){
    Object[] t = null;
    for(Object[] z : nodes)
      if(z[0] == x){t = z; break;}
    if(t == null){return false;}
    for(int i = 1; i < numchildren; i++)
      if(t[i] == null){t[i] = y; return true;o}
    return false;
  }
}
```

Figure 4.2: An implementation of the `Tree` interface backed with a set.

operation (where n is the number of *non-leaf* nodes in the tree). The following query finds the tuple representing `x` in this tree structure:

```
selectAll(Object[] o : tree | o[0] == x);
```

Note that in the worst case this query iterates the whole tree structure to find the children for a given value. Once we have these children, if we wish to keep walking the tree, we must iterate the tree again for each child, to find that child's children, and so on.

An ADT can use a specialized structure to optimize its queries and updates. For example, rather than our `Set` implementation, trees can be implemented as a hash-map that maps from `Object`s to a `TreeNode` that holds information about their children. This speeds up walking the tree — we can just look each child up in the hash-map to get its children, and so on. An example implementation is shown in Figure 4.3. The relevance of this example to object querying is the observation that the `getChildren()` method is a query, and that the `Object[]` array that each `TreeNode` holds is a *cache* for the results of that query. The performance of the `Set` version could be improved by caching the result of `getChildren()` calls, so that the entire set of parent-children tuples did not have to be iterated repeatedly. Indeed, a version of the `Set` based tree that used caching would result in a map-like structure itself.

A cache such as this map needs to be maintained. Update operations on the set of parent-children tuples will render the cache out of sync. One way to deal with this is to *incrementally* update the cached results. Rather than rebuilding the map each time it is updated, we simply update the cached results in line with the changes made to the main list. This is implicitly done in the `HashMap` implementation, where each `TreeNode` is *both* the cache and the structure it is caching. This is fast and simple for its specific purpose, but we posit that using object querying, this process can be generalized and automated.

```
class HashTree implements Tree{
  private HashMap<Object,TreeNode> nodes;
  int numchildren = 4;
  public Object[] getChildren(Object x){
    return nodes.get(x).children.clone();
  }
  public boolean addNode(Object x){
    if(nodes.get(x) == null){
      nodes.put(x,new TreeNode(numchildren));
      return true;
    }
    return false;
  }
  public boolean addChild(Object x, Object y){
    return nodes.get(x).add(y);
  }
}
class TreeNode{
  public Object[] children;
  public TreeNode(int size){
    children = new Object[size];
  }
  public boolean add(Object y){
    for(int i = 0; i < children.length; i++)
      if(children[i] == null){
        children[i] = y;return true;}
    return false;
  }}
```

Figure 4.3: An implementation of the `Tree` interface that uses a `HashMap` of `TreeNode`s internally.

A limitation of this interface is that it is inflexible. For example, if we decide we need to find the *parent* node of the node that holds `x`, we either have to manually walk the tree using `getChildren()` (starting from the root node, and checking for `x` in the children of each node we reach), or make a change to the `TreeNode` class to add a parent field. This solution, however, requires changing the interface of the ADT.

4.1.2 Query-Based Interfaces

An object querying implementation of this tree would use the `Set` of tuples as before, and provide syntax for querying this set of tuples. This allows for more general abstractions, as the interface to the set becomes the queries expressible in the query language provided. Adding *caching* and *incrementalization* allows the data structure to self-optimize. Frequently used results will be cached, and updates to those results maintained incrementally. This means by simply using the structure in a manner that requires the children of each node be frequently accessed, the structure will automatically adjust to provide performance for `children()` operations close to the specialized implementation.

A major advantage of the object querying based implementation is that it does not fix the set of queries for which it is optimized. The `TreeNode` implementation's queries are fixed by the structure of the `TreeNode` objects and the hash-map, and so optimizing the `TreeNode` based implementation for other queries requires non-trivial alteration of the data structure. Finding a node's parent, as we have said, could be accommodated by extending each `TreeNode` with a `Parent` field. This provides much better performance than walking the tree to find the parent of a given `TreeNode`. Unfortunately, it also means we have to maintain the value of another field when updating. In addition, the fixed nature of the `Tree` interface limits the gain. If the `Tree` ADT has already been used, existing code which needs to find a node's parent will already have a man-

ual implementation. Reusing the extended `Tree` is also difficult. We can extend the interface and distribute the extended `Tree`, but this grows increasingly complex and unwieldy as more developers add their specific optimization for their particular `Tree` needs. Eventually a large variety of `Tree` extensions, each with different permutations of methods to accommodate varying needs may arise. What is needed, then, is a truly flexible interface.

Object querying provides that interface. When combined with caching and incrementalization, it can achieve performance close to that of a specialized implementation. Using object querying, finding the parent of a node that holds a particular value can be accomplished like so:

```
selectAll(Object[] o : Tree | o[1] == x || o[2] == x
                            || o[3] == x);
```

If this query is evaluated frequently, the query caching system will build a map from values of `x` to the appropriate results from this query — optimizing its evaluation, and making finding the parent for a given value as fast as finding the children for that value. All without alteration to the data structure or additional programmer effort.

4.2 Cache Manager

JQL's caching support has been implemented using AspectJ (Appendix A has a brief overview of AspectJ for readers unfamiliar with it). An aspect, the cache manager, intercepts all calls to evaluate queries, and supplies cached results back, if it has cached results for that query. When a query that is not being cached is evaluated, the cache manager aspect may decide to start caching that query (this is decided by a caching policy, which we explore further in Section 4.5). Each cached query is mapped to a set of cached results. Queries are identified by their source collections, the query expressions and any other variables involved. Two queries will map to the same cache if they operate over the same collections and variables, with

the same expression. Recall our example query from Section 2.2:

```
selectAll(Dev d:devs, Employee e:emps |
                      d.email.equals(e.email));
```

The results for this query will be a set of tuples of `Developer` objects and `Employee` objects, which we store in the cache manager aspect.

Storing Employee objects within the cache manager does not interfere with garbage collection. Any objects stored in a cached query result are members of a source collection, and as such there always exists a 'live' reference to them from their collection already. If the objects are removed from their source collection, they are also removed from the query cache (see Section 4.3), and can then be garbage collected without interference from the cache.

4.3 Incrementalization

Whilst caching a query can offer a large performance benefit, it is essential that cached queries return exactly the same results as they would if they were uncached. It is possible for events to occur in program execution which could render the cached query results inaccurate, and it is therefore necessary to account for these changes. A simple approach would be to invalidate the cache whenever such an event occurs; however this reduces the usefulness of the cache. JQL instead accounts for these changes by updating the cached results whenever an event happens in the program that could change the results. Two events can potentially change the results of a query that we are caching: either objects can be added to or removed from a collection that is being queried; or, one of the fields on an object that is part of one of the queried collections could be changed.

4.3.1 Addition/Removal of Objects

Adding a new object into one of the source collections for a query can mean that new results need to be returned from that query's evaluation. If one of the `Developer`s from our example query is hired by the company, and has a corresponding object added to the set of `Employee`s, a new `[Developer, Employee]` pair needs to be added to the result set. It cannot mean that fewer results need to be included, however, as each tuple's evaluation is completely independent. Updating the cache therefore only requires adding whichever new tuples this object causes to the result set. To find these tuples we use the existing query pipeline, and evaluate a simplified version of the query. This simplified version has the newly added object substituted in place of the domain variable that is bound to the collection it has been added to.

Object removal is much the same — it can only cause the removal, not addition, of tuples to the results. We proceed in much the same fashion, evaluating a simplified query pipeline, but instead of adding the results to the cache, we remove any we do find in the cache.

We determine when these update events are necessary using AspectJ, which allows us to instrument collection operations that alter a collection's membership, such as `Collection.add(Object)`. When such an operation occurs, the cache manager checks the return value of the operation (`Collection` operations that modify the collection are required to return `true` when they successfully alter the collection they are called on), verifies that the collection is involved in a cached query, and then updates the caches for any queries in which this collection is used.

4.3.2 Object State Updates

Changing the state of objects that are involved in a query can also affect the consistency of the cache. If a query expression makes use of the value of a particular field, updates to that field could mean that an object no longer

passes the query expression (or vice versa). We therefore need to check for all updates to fields on objects in any source collections. To avoid having to instrument all field writes in a program and then check each field write for its impact on cached queries, we have introduced an annotation on fields: `@Cachable`, which the JQL frontend adds to fields that are used in query expressions. The cache manager aspect will only advise field updates on `@Cachable` fields. It also checks when deciding to cache a query if that query uses any fields that are not marked `@Cachable`, and marks that query as uncachable if it does. This can arise when the source for objects used in a query has not been processed by the JQL front-end. Even using this annotation does not completely remove the problem of unnecessary checks on field sets. Field sets on a `@Cachable` field on an object not contained within a queried collection will incur overhead, as the incrementalizer must check to see if the field set has influenced any queries. The `@Cachable` annotation is an unfortunate implementation detail of our prototype; an ideal implementation that was more strongly integrated into the language would dispense with it.

4.4 Caveats

Collection Implementations

JQL's caching system is not foolproof. It assumes that collections implementing the `Collection` interface adhere to the interface specifications. It is particularly important that collections correctly return a value to indicate whether or not a collection operation altered the collection. Whilst these facets of the specification are not (and cannot) be explicitly defined in the interface itself, they are part of the accompanying Javadocs. It would be simple to break the incrementalization by implementing `Collection` in a class that returned `true` for its `add` method, but that did nothing at all. This would fool the incrementalizer into adding results for whichever

object was passed to `add`. Practical reasons for implementing such a collection are, however, beyond us. For the standard `Collections` library, and extensions derived according to the specification, the JQL prototype functions correctly.

Method Calls

Another problem is the use of method calls within the query predicate (in particular `get()` method calls), but any method call dependent on the state of the object it is being called on. If a `get()` method call is used to check the value of a field, instead of direct public access, the cache manager cannot know that updates to that field could cause the cache to become inconsistent. At present, the use of method calls within a query expression cause that query to become uncachable. Possible solutions to this problem could include requiring that all fields on any object used in a query, or used by any method call in a query, be marked `@Cachable`, and update the cache on writes to these fields. Another solution is to extend the `@Cachable` annotation for method calls, perhaps to allow the specification of fields that this method call's return value depends. Unfortunately these solutions require either the developer manually specifying fields used, or analysis of method calls used in queries to find the fields used.

Synchronization

A final caveat is synchronization. Our JQL prototype at present is unsynchronized. This means, in particular, that an object can be added to a collection by one thread, triggering a cache update, whilst another thread queries that collection. The thread querying the collection may not have returned to it any of the additional results from the newly added object — although, as the cache update takes place within the `add()` call (before its return value is passed back to the program), the object addition can be

considered not to have finished until the query cache has updated in any case.

4.5 Caching Policy

Whilst incrementalization does improve the usefulness of caching, by allowing the cache to stay valid even in the presence of program state updates, maintaining the cache incurs overhead for the rest of the program. If a query is very infrequently evaluated, but its cache is frequently updated, the overhead of maintaining that cache can outweigh the benefit of caching the query in the first place. Caching policies are heuristics to determine if a query should be cached, or if a cached query should have its caching suspended. The performance characteristics of a cached querying system can be quite complex, and are highly dependent on the execution profile of the actual program being run. If there is never any need to update the cache, then there is no overhead and caching should always be used. Adding updates then requires the caching policy to balance the overhead of keeping the cache updated against the benefit of having the cache. This requires the policy to estimate how much updates, evaluations, and cache-building all cost, as well as roughly how many updates we can expect to be doing per-query. Accurately estimating these costs is impossible, and so heuristics are used by the caching policy to determine when to start or stop caching a particular query. Our JQL prototype includes a few simple caching policies. There is a naive, 'always-cache' policy; a complexity based policy and a timing-based policy.

Naive Policy

The Naive Policy starts caching every query as soon as they are first evaluated, and never stops caching them no matter how numerous or costly the updates for a query are. As the naive policy is so simple, it does not

require keeping track of any statistics for query evaluation.

Complexity Policy

The Complexity Policy checks the number of consecutive updates (that is, updates to a query cache without an evaluation) against a certain threshold. This threshold is determined by how complex the query is — in this case, complexity is mostly determined by how many domain variables it has. The more source collections a query has, the more costly updates to the query cache are, as the new object must be tested against all possible combinations of objects from the other source collections. If all collections are of the same size, the cost is $O(n^{k-1})$, where k is the number of source collections. The threshold also takes into account the size of the query, where the size is the product of the sizes of the input sets. Once the threshold for consecutive updates has been reached, the query's caching is halted until an evaluation occurs, at which point the query is re-cached.

Timing Policy

The Timing Policy tracks the average time to evaluate each query (uncached, and a separate average time for cached queries), the average time taken for each update to this query's cache, and how long has been spent updating this query since it was last evaluated. When the time spent dealing with consecutive updates to this query's cache exceeds one quarter of the query's uncached evaluation time, the caching is halted until an evaluation occurs.

These policies are quite simple, and research into more advanced policies would be useful future work for JQL.

4.6 Performance

Caching can provide a large benefit to the performance of programs. However, it does introduce overhead — the cache must be built, and from then on must be kept consistent with changes in program state. We present three sets of experiments, to investigate the trade-off between cached and uncached querying. In the first set we investigate using caching to query a Graph ADT (very similar to the situation outlined in Section 4.1.1), and compare it to more conventional approaches. In the second set of experiments we investigate the performance impact of maintaining the cache while updates occur versus the gains provided by the caching.

In all experiments which follow, the experimental machine was an Intel Pentium IV 3.2GHz, with 1.5GB RAM running NetBSD v4.99.9. In each case, Sun's Java 1.5.0 (J2SE 5.0) Runtime Environment and AspectJ version 1.5.3 were used. The standard `System.currentTimeMillis()` method was used to perform timing. The source code for the JQL system and the query and caching benchmarks used below can be obtained from `http://www.mcs.vuw.ac.nz/~darren/JQL/`.

4.6.1 Flexible View

The purpose of this study was to investigate the performance actually offered by the JQL prototype, compared with a hand-coded caching implementation. The benchmark queries a Graph ADT, similar in spirit to the Tree ADT outlined in Section 4.1.1. The graph was represented as a set of pairs of nodes, with each pair indicating an edge between those two nodes. The benchmark finds the shortest path from every node in the graph to every other node in the graph, using Djikstra's Shortest Path algorithm [11]. This algorithm relies on finding the neighbours of a given node in the graph, and our experiment investigates three different methods for finding them:

- **Manual Cache** — First constructs a hashtable which maps from a Node to its neighbours. When the shortest path algorithm needs the neighbours of node `n`, it can look them up in this map. This corresponds to manual caching, and to an adjacency list implementation.
- **Query** — Locates the neighbours of a node `n` using the following query:

  ```
  neighbours=selectAll(Edge e:Edges| e.head == n);
  ```

  This query returns the set of all edges which contain `n` in their head field.

We tested the query implementation with caching enabled and disabled, to compare the two.

Experimental Setup

All implementations were tested over the same set of randomly generated graphs. Each graph consisted of 103 nodes, with varying number of edges generated, from 100 edges to 15,000 edges. The graph is generated before timing starts. Our figures for the running time of each execution of the benchmark are the average of 15 runs of the benchmark, with all query caches and stored hashtables cleared before each run. These parameters were sufficient to generate data with a variation coefficient of ≤ 0.15 — indicating low variance between runs.

Discussion

The results of the experiment are shown in Figure 4.4. The main observation to be made here is that the query, when cached, follows a very similar performance curve to that of the manually cached. When uncached the query increases extremely steeply with the size of the set of edges. This is

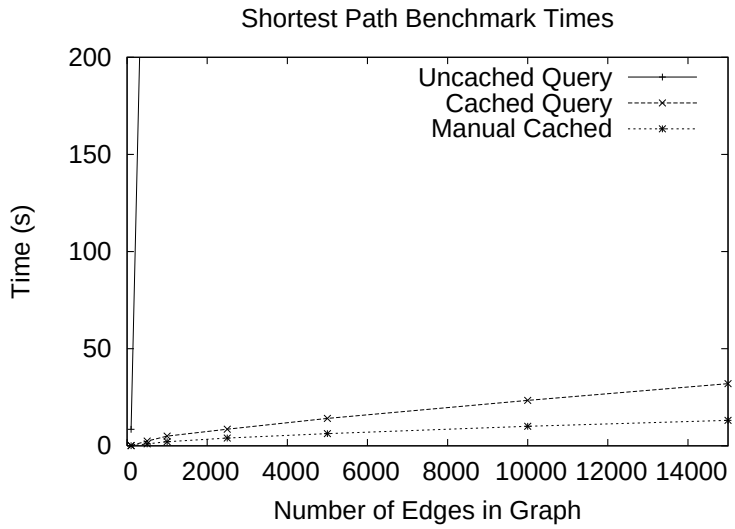


Figure 4.4: Experimental results comparing the evaluation time of a shortest path benchmark over graphs with varying numbers of edges.

because once the query has been used to find a node's neighbours, those results are cached by the cache manager. Once every node has had its results cached, finding the neighbour of any node is effectively looking it up in a hash map — which is just what is done by the Manual benchmark. The difference in time taken between the Manual and the (cached) Query implementations can be accounted for by overheads, such as the overhead associated with instrumenting methods using AspectJ, and the cost of evaluating the query before it is cached (which happens once per node).

4.6.2 Cache Updating

Augmenting queries with caching greatly improves the time it takes to evaluate them. However, once a query has had its results cached, it must then be kept up to date with changes in the program state that may affect these cached results. The JQL prototype has support for incrementally updating the cache to maintain the correctness of its cached results, and

we have conducted some experiments to explore the trade-off of querying performance against cache update overhead.

There are two ways that query results can be altered between evaluations — either objects can be added to or removed from one of the source `Collections` for a query, or the value of a field on an object that is checked in the query can be changed. In either case, it is necessary to update the cached results to reflect what would be returned if the query were actually evaluated at that time.

Experimental Setup

We have constructed two sets of tests to examine the performance benefit and overhead related to caching with incrementalization. The first set of tests examines the cost of keeping a cache up to date when objects are being added to the query's source collections, and the second set examines the cost of maintaining the cache when objects within the query's sources are being changed in a manner that could affect the results of the query. For both sets of tests, we use `Student` and `Course` objects, and link them with `Attends` objects which denote that a `Student` attends a certain `Course`. Both sets of tests run the same queries, given in Table 4.1

For the object addition benchmarks, the experiment first generates 500 `Student` objects, and then loops 500 times, adding a new `Attends` object to the `attendances` list, and possibly doing one or more query evaluations, depending on the experimental parameters. As `attendances` is part of a query, each addition to it causes the query cache to be incrementally updated. This benchmark is run with one parameter: the ratio of queries to updates.

The object alteration benchmarks are similar: the experiment first generates 500 `Student` objects, and adds 500 `Attends` objects as well, before any queries are called. It then loops 500 times, changing the `course` field in an `Attends` object, and possibly doing one or more query evaluations, depending on the experimental parameters. As `course` is a field that is

Name	Details
`OneStage`	`selectAll(Attends a:attendances | a.course == COMP101);` This query has a single join, and a single domain variable (COMP101 is a constant value for a Course object). Updates to this query requiring checking each affected object's course field against COMP101.
`TwoStage`	`selectAll(Attends a:attendances, Student s:students| a.course == COMP101 && a.student == s);` This benchmark requires two pipeline stages and has two domain variables. Updates to this query require checking each affected object's course against COMP101, and comparing the object's student field against all student objects in students.
`ThreeStage`	`selectAll(Attends a:attendances, Student s:students, Student t:students| a.course == COMP101 && a.student == s && t.id < s.id);` This benchmark requires three pipeline stages, and has three domain variables. Updates to this query require comparing an updated object with all students, and then comparing those results with all students again.

Table 4.1: Details of the three benchmark queries

used in the query, each alteration of it requires that the cache be updated. This benchmark is also run with one parameter, the ratio of queries to updates.

Discussion

Graphs for the object addition experiments are shown in Figure 4.5, and graphs for the object alteration benchmarks are shown in Figure 4.6. The x-axis denotes the ratio of queries to updates, and the y-axis the time taken for the whole benchmark to execute.

The main point of interest in these graphs is the 'cross-over' point, where the uncached line crosses over the cached line. This is where the benefits of caching begin to outweigh the overheads incurred. The main factor determining where this point is is the complexity of the query itself, specifically how many source collections there are. As the complexity increases, the cross-over point shifts to the right — that is, proportionally more queries need to be done to make up for the cost of maintaining the cache. For the object addition test, the crossover point for the `OneStage` benchmark appears at approximately ten queries evaluated (over five hundred updates). This means if we do at least one query for every fifty updates, caching that query will pay off. Single variable queries are cheap to update, as they only require checking the new object against the query predicate. As we discovered in section 2.1, operations over a single collection are by far the most common, and so these results are encouraging for the usefulness of incrementalized querying for collection operations.

By comparison, the performance for queries with more than one domain variable is more complicated, and highly dependent on the specific query. For the `TwoStage` query, the cross-over point at which caching pays off occurs at approximately 125 queries, or one query for every four updates. For the `ThreeStage` query, caching doesn't pay off until the number of queries hits about 1600, or more than three queries for every

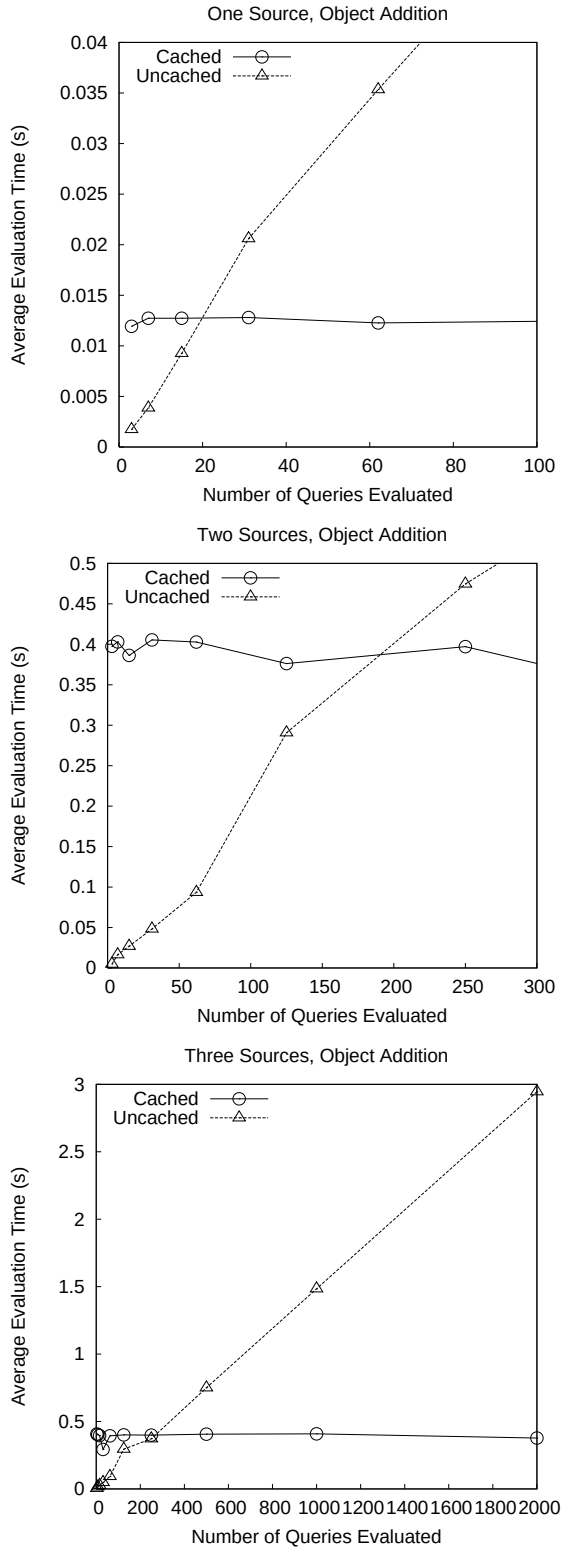


Figure 4.5: Experimental results comparing the evaluation time with and without caching enabled, for the object addition benchmarks. Data for the OneStage, TwoStage and ThreeStage benchmarks are shown at the Top, Middle and Bottom respectively.

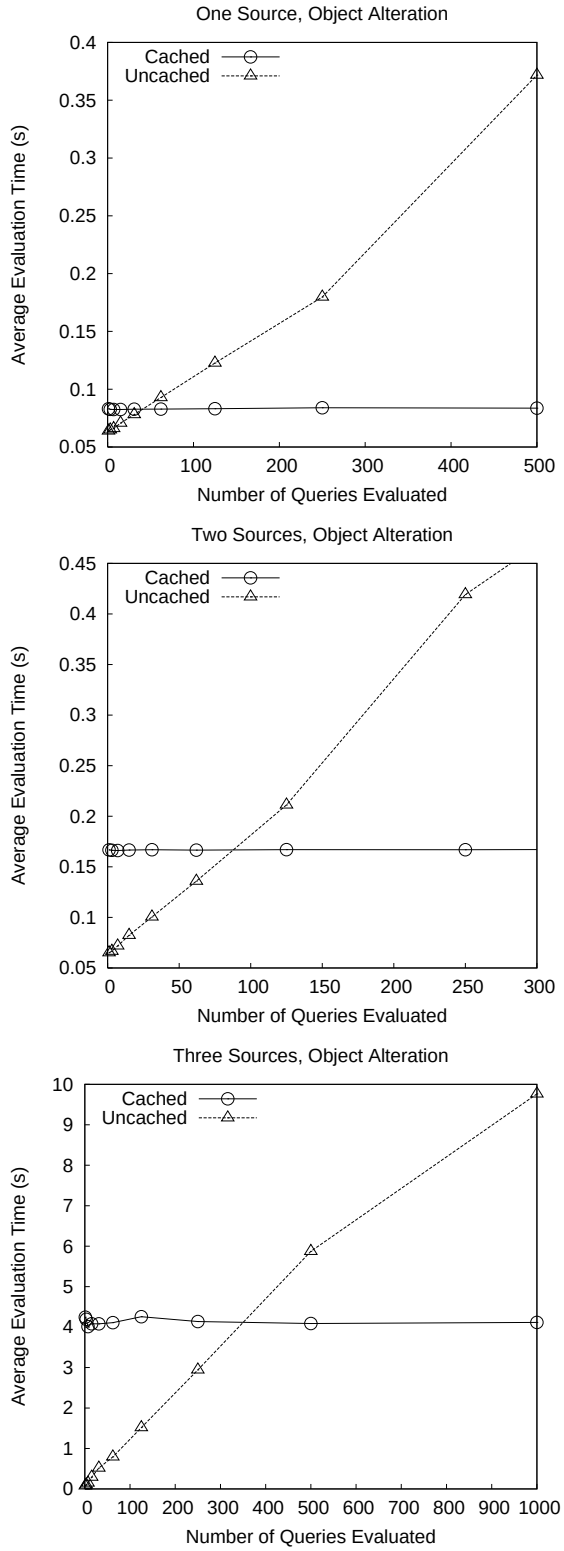


Figure 4.6: Experimental results comparing the evaluation time with and without caching enabled, for the object alteration benchmarks. Data for the OneStage, TwoStage and ThreeStage benchmarks are shown at the Top, Middle and Bottom respectively.

update.

Single variable queries will always only require one evaluation of the query predicate to update, multiple variable queries can take more. At worst, they can require the predicate to be evaluated a number of times equal to the product of the sizes of all the input collections except for the one the updated object is a member of. For example, every `Attends` object added for the `TwoStage` benchmark needs to be compared against every `Student` object in the collection `students` (five hundred evaluations total). In the `ThreeStage` benchmark it needs to be compared against every `Student` in `students`, which are then compared with every `Student` in `students` again. This requires, potentially, 250,000 evaluations per update (while this is high, consider that a full evaluation of this query could require up to 125,000,000 evaluations). These are, however, worst case figures, and it is likely that the query optimizer will be able to alleviate these costs by using appropriate join strategies and querying order.

The object alteration benchmarks show the extra overhead when the updates are not just dealing with a new object, but instead dealing with a change to an object's field when that field is used in the query expression. This has effectively double the overhead compared to adding a new object. First, the cache manager has to remove any old tuples that depend on the previous value of the field that was changed, as we cannot guarantee that the query predicate holds for these tuples anymore. We then update the cache as though we had just added the changed object.

These experiments help outline the kind of performance that can be expected in general for query caching, but it is important to stress that the performance for queries can vary quite markedly depending on the actual query predicate being evaluated, and the optimization possibilities it offers. We explored this in our experiments in the previous chapter, and the join ordering techniques there apply to the 'subqueries' used for cache updates as well.

Chapter 5

Related Work

5.1 Loop Analysis

The observation that loop constructs are most frequently used for operations on data structures has been made before. Pratt [41] categorized loops in a Pascal2 compiler, according to their intent (such as ”Search elements of a vector” or ”Generate a geometric progression”), and tabulated them according to the actual control structure (`while`, `repeat` or `for`) used to implement the loop. He found correlation between the structure a loop operates over and the syntax used to implement that loop. `while` loops were typically used for searching linked lists, for example, whereas `for` loops were typically used for Pascal's native vectors. He also noted that most loops follow one of a few general patterns, and that these patterns are usually related to the processing or searching of data structures used in the program. Much as we do, he proposes additional constructs to support more abstract operations over data structures — indeed, he even suggests a `first` construct that searches a collection for the first object for which a boolean expression holds.

Other work that categorizes or analyses loops does so with an eye to improving program analysis. Waters [46] categorizes loops into loop plans, using 'Plan Building Methods' such as augmentation (much like re-

duce), filtering, control and interleaving (where one loop does work that can really be seen to be two loops). These plans can then be used in a logical proof of the correctness of the analyzed program, and a tool was created to produce such plans.

Abd-El-Hafix and Basili outline another taxonomy [1], with a similar method for analyzing loops, and a similar goal of improving program analysis. Their taxonomy is based on the characteristics of a loop (such as the complexity of the control condition), and again produces plans that can be used with predicate logic. Their system works with a knowledge base of plans, matching program code up with pre-defined stereotypes. Both of these studies analyzed loops based on the code that makes up the loop body and how the loop's conditions are expressed, as opposed to categorizing based on the intent of the loop, as we presented in Chapter 1.

Soloway *et al.* conducted an empirical study on looping constructs and their impact on solving programming problems [43]. Groups of students were given a simple programming task, and one of two variants of Pascal to use to solve it. The variants of Pascal provided different syntax for loop control structures, one of which corresponded easily with most student's preferred solution to the problem, and the other which did not. They showed that, especially for novice programmers, the available control syntax greatly affected the correctness of the resulting program.

Less explicitly focused on loop analysis, but still related, is the idea of *cliche* based programming [47], where tools support the programmer by providing implementations of a wide variety of so-called cliches. These cliches are operations that programs are commonly built out of — such as finding the shortest path through a graph, or searching a collection for a certain value. Our loop categories from Section 2.1 are close to cliches. Like queries, the use of cliches focuses more on the intent of the code, rather than the explicit details of its execution.

5.2 Incrementalization

Incrementalization for program optimization has been explored by Liu *et al.* in a series of papers, which explored and developed methods for automatically analyzing and transforming programs to incrementally evaluate various expressions. They outline a three-stage analysis for converting a general computation to an incrementalized version, which they call *cache and prune* [35]. Their analysis identifies the incremental steps and useful working values for a given computation, which can be cached. They also describe a systematic method to transform a program to use these values [34]. The main point of difference between their incrementalization techniques and those used in JQL is that their system uses program transformation to statically convert a program from a 'naive' implementation to an incrementalized one. Their incrementalizations cannot be deactivated on the fly to cope with situations where the incrementalization overhead outweighs the benefits.

CACHET, a prototype implementation based on the cache and prune technique, is described in [29]. CACHET is an interactive program transformation system, which operates on a first order functional programming language. Developers manually select expressions to transform and can view the transformations as they are applied.

The performance benefits of incrementalization are explored for optimizing recursion [30]. They present a general method for converting recursive operations into iterative operations, using incrementalization. Similar to the cache and prune method, it uses three stages: identifying base cases, identifying increments, and transforming the program. A case study of optimizing Ackermann's Function, a complex recursive function, using their approach shows much improved space and time complexity [31], demonstrating the value of incrementalization.

They also use incrementalization to optimize aggregate array computations [33]. These are much like the *reduce* operation described in Section

2.1 — they compute an accumulated result over an array. Often these results depend on overlapping elements in the array, leading to redundant computations. They present algorithms to identify these computations, transform them into incremental evaluations, apply their cache and prune strategy to maintain intermediate values, and finally form the new loop statements to complete the transformation.

Incrementalization for use in object-oriented programming is explicitly addressed in [32]. This presents the view that abstract data types are really definitions of a set of allowed queries and update operations. They then present a method for deriving incrementalized versions of expensive operations, in an object-oriented context, using set comprehensions as an example. They also describe the idea of a *library* of incrementalization rules. The library would be an expandable set of rules to identify and incrementalize expensive operations used in object oriented programming. They also show performance data from a prototype system, implemented using Python, which automatically incrementalizes other Python applications. Again, this system uses compile-time program transformation.

A theme of these works is that incrementalization, in general, leads to larger and more complicated programs, with much better performance. The aim, then, is to allow programmers to write straightforward, simple implementations of their code (specifying *what* they want to do) and leave the optimization (the *how*, be it incrementalization or some other implementation) up to the compiler.

5.3 Query Based Debuggers

5.3.1 Instrumenting Debuggers

Lencevicius *et al.* have developed a series of *Query Based Debuggers*. These are debugging tools that operate using queries over objects in a running program, and were designed to solve the problem of finding incorrect ob-

ject relationships in the large, complicated tangle that is the object graph of a program. Their first debugger, implemented for Self [44], is described in [25]. This debugger was designed to allow the specification of complex invariants, and so required a modification to the Self virtual machine to allow for querying over **all** objects of a specified type. Conceptually its querying operations are quite similar to JQL's — it also uses hash joins to evaluate queries when possible (it lacks a sort join, however), and uses similar join ordering strategies to JQL.

This was followed with the *Dynamic* Query Based Debugger, implemented for Java [26, 27, 28]. The dynamic query based debugger addresses the *cause-effect gap* [13]. This is the gap in time between when an incorrect program state occurs and when it becomes a noticeable problem for the system. For example, the gap between when an illegal (but unchecked) assignment in a tree structure occurs, and the errors that arise when the next attempt is made to walk that tree structure.

The dynamic query based debugger evaluates queries just like the original query based debugger, but it also ensures that the query expression holds at all times during the program's execution. It uses incremental evaluation, coupled with a custom classloader that modifies bytecode so that field sets are instrumented with debugging code, to allow checking of the query whenever its results could be changed. The debugger would then notify the programmer whenever a query invariant had been violated, immediately after it happened.

In order to check its invariants against all instances of a certain class, the debugging code also instruments constructor calls, and keeps references in domain stores of all objects created during the program execution. This does not use `WeakReferences`, which means it interferes with garbage collection (in fact, no objects in the query domain will be collected at all). An interesting optimization used by this debugger is the replacement of simple, one-variable queries ('selection' queries) with a compiled bytecode version of the query expression, generated by the debugger for

that particular query.

Finally, their On-The-Fly Query-Based Debugger was an extension of the dynamic debugger [24]. The dynamic debugger required that queries be specified before any classes were loaded, to allow their custom class-loader to instrument any events that could possibly impact a query. To allow for interactive query specification at run-time, the On-The-Fly debugger instruments all operations that could potentially impact *any* query: set operations on object fields, and all constructors. While very flexible, this approach unfortunately incurs execution overhead, even if the debugger is disabled, while instrumented code is used.

These debuggers all have querying systems quite similar to JQL's. However, they do not pass their results back to the program for use by the programmer; nor do they aim to. Queries for these debuggers are usually phrased as expressions where the desired outcome is no result at all, indicating that no objects fail a set of constraints, and raising an error when results are returned.

Hobatr and Malloy have implemented a query-based debugger for C++ [19, 20]. This operates in a manner quite similar to the dynamic query based debugger. Queries are specified using the Object Constraint Language, and are compiled by the frontend into a form usable by the debugger's backend. The backend generates code using the OpenC++ Meta-Object Protocol, which can then be compiled into C++ source, which can then be compiled and executed. This final executable will have the query based debugging code built-in, and will terminate and report an error if any constraints are violated. This debugger handles pre/post conditions for methods, and class invariants. Queries cannot be specified at run-time. While the capabilities (and performance characteristics) of this debugger are unclear, it seems unlikely that it allows 'join' queries.

5.3.2 Static Debuggers

The FOX is another query based debugging system for Java, which takes a completely different approach [40]. Rather than instrumenting a Java program, the FOX operates on heap dumps. Heap dumps are snapshots of the state of all objects in a running Java program (in Java, all objects are created on the heap). The FOX uses the HAT (Heap Analysis Tool) [14] to generate its heap dumps. The FOX was mainly created to focus on object aliasing and ownership constraints, and its query syntax is difficult for use in other areas.

Terrier is a more general-purpose descendant of The FOX, and also operates over heap snapshots [3]. Terrier uses BeanShell [39] to evaluate query expressions (which are written in Java), and returns results via a webserver. Heap snapshot based systems operate almost entirely independently of the debugged program — they require no instrumentation or custom JVMs, and incur no overhead. They do require that the heap be dumped (this dumping can be invoked programatically), which makes them more difficult to use effectively, as the exact timing of the heap dump can be of crucial importance. As they are operating on 'dead' objects, they cannot invoke methods in their query expressions, which limits the range of potential queries severely.

5.3.3 Program Trace Debuggers

Goldsmith *et al.* present the Program Trace Query Language (PTQL), and PARTIQLE, a compiler to support it [15]. PTQL supports relational queries over program execution traces. PTQL is very similar to SQL in syntax, making use of `WHERE`, `SELECT` and `FROM` clauses. Unlike the previous query based debuggers, and unlike JQL, PARTIQLE does not run queries over objects. Instead, it runs queries over traces of program execution, including timing information for program events. It is still an online debugger, however — these traces are built at runtime, and the queries eval-

uated at runtime as well. PTQL's focus is different from other query based debuggers, which investigate the relationships between objects. Instead, PTQL is for specifying queries over a program's *behaviour*. An interesting example given is a query to find repeated `String` concatenation operations.

Similar to PTQL is PQL, the Program Query Language [36]. PQL also allows querying of the sequence of events over objects in a program. The PQL system generates static checkers for the specified queries, and also has a dynamic component which makes use of these static checkers to instrument as little of the program as possible. PQL's syntax involves specifying types of objects to check for, and behaviour of those objects to match. PQL queries can also specify a *lack* of a certain behaviour within a certain scope. An interesting ability of PQL that other querying systems lack is the ability to dynamically replace matched events with a 'corrected' version; an example is replacing an unsafe SQL call to a database with a checked SQL call.

5.4 Object Querying Systems

Particularly similar to JQL is the recent development of Microsoft's Language INtegrated Query (LINQ) project [37]. LINQ aims to add first class querying support to .NET languages (in particular Visual Basic .NET and C# [38]). LINQ operates by translating queries into additional methods on collections of objects, which then perform filtering and mapping on the collection. LINQ's scope is wider than JQL's, providing integrated querying for object collections, XML structures and SQL databases. It is unclear at present what optimizations LINQ provides for object querying, or if it provides (or is planned to provide) incrementalized caching. Cω [6, 7] is the research language from which LINQ has developed.

Python features list comprehension expressions that allow for query-like expressions over collections of objects. These expressions always re-

turn lists (list generators are also supported, that return lazy-evaluating iterators). List comprehensions can filter collections and map collections. They can be specified over multiple collections and nested. A common use for these list comprehensions in Python is to pack and unpack Python's native tuple objects. Much like LINQ, it is unclear what optimizations, if any, Python's list comprehensions provide, especially in situations with multiple source collections and complicated filter expressions.

Languages that support 'block' style constructs that can be used as parameters to methods (sometimes known as 'lambdas', 'delegates', etc.) can support some of the loop operations categorized in Section 2.1. In the Ruby standard library, for example, the `Array` class features a `select` operation which takes a block and returns a collection of all elements of the `Array` for which the block evaluated to `true`. `Reduce` is also possible (termed 'inject' in Ruby), and `Map` is trivial. Operations like this do not usually provide for efficient joins between collections, or dynamic ordering of query stages.

5.5 Databases

Cook and Rai present Safe Query Objects [10], a system allowing compile-time translation from a representation of a query as objects to a Java Data Objects (JDO) [42] query. JDO queries are usually written as strings within the program, with arguments interpolated and conditions concatenated, and so on. Unfortunately this means the queries are not checked for correctness at compile-time — a typo in the JDO query string will be missed by the compiler and cause an error at run-time instead. They are also more difficult to verify for safety. Safe query objects allow the compiler to take care of these issues.

Object Oriented Database Management Systems are similar to relational database management systems, however they provide additional, object-oriented facilities. These facilities, as outlined in [4], include encap-

sulation, inheritance, and other OO mainstays. These systems do allow for objects to be queried, however they differ from our approach as objects must be in the database to be queried. This database is usually a separate process, or even on a different computer entirely. Object Oriented Database Management Systems also provide persistence of objects, which JQL does not provide. These databases are typically queried with the Object Query Language (OQL), which resembles a subset of SQL [2].

Query optimizations for databases, relational and object-oriented, are a large area of study ([21] and [9] are two surveys of the field). Optimizers attempt to find the most optimal query evaluation plan that is within the space of possible plans. They also frequently use sampling techniques to analyze selectivities of joins (e.g. [45], [5]). Dynamic programming techniques such as memoization are often used as well, to avoid redundant computations within the query optimizer. For object oriented databases, the query optimizer may also need to take into account expensive method calls, which can render even single-sourced queries more expensive than join operations [18]. JQL at present does not account for expensive method calls.

Database systems can incrementally update cached queries (or *materialized views*, as they are termed). The decision to incrementally maintain a view is typically made manually by a database administrator, instead of automatically by the query system. Algorithms for incrementally maintaining these results are described in [17]; the basic concept of evaluating just tuples which have changed, and merging these with the stored results is the same. These algorithms also sometimes operate in situations where not all source tables are available, limiting the operations which can be incrementally updated. When removing tuples from the materialized views, these algorithms can use a form of reference counting to determine which tuples are affected. They count the number of ways that a tuple can enter the result set, and use this information to more efficiently update the cached view. This approach does not apply to JQL, as each tuple within

the result set has only one derivation.

5.6 Multi-Index ADTs

The idea of ADTs with a more flexible interface is similar to the concept of multi-index collections, such as those offered by the Boost C++ Library [8]. These are collections implemented using templates to provide multiple ways to access items — a typical example is sequential access to items in the collection, but with different indices corresponding to different orderings of the items. Multi-Index collections effectively allow for the specification of several different interfaces to the data contained within the collection. The interface used to query the collection then becomes a parameter to methods that access the collection. These indices have to be specified at compile-time, however, so they cannot be enabled and disabled, and they are manually specified by programmers, rather than being automatically generated by a caching system.

Chapter 6

Conclusions

6.1 Summary

In this thesis, we have presented JQL, the Java Query Language, an extension to Java to provide first-class support for object querying. Using JQL provides higher level abstractions of collection operations, and allows developers to take advantage of automatic, intelligent optimization of their queries.

First, we outlined the basic concept of object querying, and some situations where it may be useful. In particular, the benefits of dynamic reconfiguration of query evaluation ordering were explained. We also presented a small survey into the use of loop structures in existing Java programs, and the intentions behind these loops.

JQL's implementation was described. We showed how JQL breaks a query into stages, and the heuristics it uses to order these stages as intelligently as it can. We also showed the techniques JQL uses to optimize particular joins: the Hash Join and the Sort Join. Performance data was also presented showing the value of these optimized joins, and of the automatic join ordering system.

We then described the caching and incrementalization system added to JQL, and the idea of using object querying to implement abstract data

types with highly flexible interfaces was presented. We outlined how we use AspectJ to instrument Java programs to replace query evaluations with cache retrievals. We also showed the use of AspectJ to intercept events that require cache updates, and deal with them accordingly. We then presented experimental data showing the practicality of using cached object querying for implementing flexible data structures. Performance data pertaining to incrementalization overhead, and the effects of various caching policies, was also presented.

6.2 Contributions

Our main contribution is the design and implementation of JQL, a prototype object querying system that allows developers to easily write object queries in a Java-like language. This prototype provides automatic optimization of operations that join multiple collections together. We have presented heuristics for dynamically configuring query evaluation, and shown their relative merit. We have demonstrated this prototype's efficiency and practicality, especially when compared to manual implementations from all but the most careful developers.

We have implemented an incrementalized caching system on top of this object querying, using aspect oriented programming. We have again demonstrated the efficiency and practicality of the enhanced JQL system, and investigated the performance gains and overheads incurred by using this caching. We have presented caching policy agents, which attempt to mitigate the impact of incrementalization, and outlined some simple ideas for how these agents work. We have shown the usefulness of this incrementally cached object querying system for implementing flexible data types, while attaining performance comparable to using data types with special structures optimized for a fixed, limited interface.

We have also presented an investigation into the usage of loop control structures within 'real-world' Java programs. This showed, firstly, that the

majority of loops are over collections of data, and secondly, that most of these loops over collections fit one of a few broad patterns. We also have shown how querying could or could not be useful for the most prominent patterns of these operations.

6.3 Future Work

The most compelling future direction for JQL is to attempt to integrate it fully into the Java language, a task that will hopefully be simplified with the recent opening of Java. This should allow for the compilation of query expressions, and more useful type information for returned object tuples.

While JQL aims to provide automatic selection of caching policies and selectivity heuristics, it would be desirable for developers to be able to tweak this selection. JQL's syntax does not provide for this at present; optional 'Java-like' syntax for these capabilities would be helpful. Tool support for JQL is a high priority as well.

The hardest problems remaining are finding optimal join orderings, and good cache policies. Our present 'exhaustive' join ordering heuristic is very good but it is fallible. Caching policies are even more complex; there may even be value in static program analysis to identify updates and evaluations, to generate caching policies tailored for individual queries.

Finally, we'd like to investigate uses for JQL further. We have investigated using JQL for a tree data type in Chapter 4; we would like to find other data types for which querying could be useful. We would also like to investigate further the difference in the structure of programs that are created when developers have a cheap collection joining operation — we noted in our study into loop uses in Chapter 2 that developers rarely use nested loops. We believe this could lead to programs with less information encoded into the structure of their data, with more emphasis placed on how the querying system presents the data to the developer.

Bibliography

[1] ABD-EL-HAFIZ, S. K., AND BASILI, V. R. A knowledge-based approach to the analysis of loops. *IEEE Transactions on Software Engineering 22*, 5 (1996).

[2] ALASHQUR, A. M., SU, S. Y. W., AND LAM, H. OQL: A query language for manipulating object-oriented databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands* (1989), P. M. G. Apers and G. Wiederhold, Eds., Morgan Kaufmann, pp. 433–442.

[3] ANDREAE, C., GORDON, D., POTANIN, A., NOBLE, J., AND BIDDLE, R. Terrier: Static query-based debugging in Eclipse. Object Oriented Programming Systems, Languages and Applications Poster Session, 2004.

[4] ATKINSON, M., BANCILHON, F., DEWITT, D., DITTRICH, K., MAIER, D., AND ZDONIK, S. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases* (Kyoto, Japan, 1989), pp. 223–240.

[5] BABCOCK, B., AND CHAUDHURI, S. Towards a robust query optimizer: a principled and practical approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2005), ACM Press, pp. 119–130.

[6] BIERMAN, G., MEIJER, E., AND SCHULTE, W. The essence of data access in Cω. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (2005), vol. 3586 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 287–311.

[7] BIERMAN, G., AND WREN, A. First-class relationships in an object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (2005), vol. 3586 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 262–286.

[8] Boost C++ libraries, http://www.boost.org.

[9] CHAUDHURI, S. An overview of query optimization in relational systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 1998), ACM Press, pp. 34–43.

[10] COOK, W. R., AND RAI, S. Safe Query Objects: Statically typed objects as remotely executable queries. In *Proceedings of the International Conference on Software Engineering (ICSE)* (2005), IEEE Computer Society Press, pp. 97–106.

[11] DIJKSTRA, E. W. A note on two problems in connection with graphs. *Numerische Mathematik 1* (1959).

[12] DIJKSTRA, E. W. The humble programmer. *Communications of the ACM 15*, 10 (1972).

[13] EISENSTADT, M. My hairiest bug war stories. *Communications of the ACM 40*, 4 (1997).

[14] FOOTE, B. Heap analysis tool. `http://java.sun.com/people/billf/heap/`, 2002.

[15] GOLDSMITH, S., O'CALLAHAN, R., AND AIKEN, A. Relational queries over program traces. In *Proceedings of the Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 2005), ACM Press, pp. 385–402.

[16] A BNF Java grammar. http://cui.unige.ch/db-research/Enseignement/analyseinfo/JAVA/.

[17] GUPTA, A., AND MUMICK, I. S. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing 18*, 2 (1995).

[18] HELLERSTEIN, J. M. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems 23*, 2 (1998).

[19] HOBATR, C., AND MALLOY, B. A. The design of an OCL query-based debugger for C++. In *Proceedings of the ACM Symposium on Applied Computing (SAC)* (2001), ACM Press, pp. 658–662.

[20] HOBATR, C., AND MALLOY, B. A. Using OCL-queries for debugging C++. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE)* (2001), IEEE Computer Society Press, pp. 839–840.

[21] JARKE, M., AND KOCH, J. Query optimization in database systems. *ACM Computing Surveys 16*, 2 (1984).

[22] KICZALES, G., HILSDALE, E., JUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. An overview of AspectJ. In *Proceedings of ECOOP* (2001).

[23] KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J., AND IRWIN, J. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 1997, pp. 220–242.

[24] LENCEVICIUS, R. On-the-fly query-based debugging with examples. In *Proceedings of the International Workshop on Automated Debugging (AADEBUG)* (2000).

[25] LENCEVICIUS, R., HÖLZLE, U., AND SINGH, A. K. Query-based debugging of object-oriented programs. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* (1997), ACM Press, pp. 304–317.

[26] LENCEVICIUS, R., HÖLZLE, U., AND SINGH, A. K. Dynamic query-based debugging. *Lecture Notes in Computer Science 1628* (1999).

[27] LENCEVICIUS, R., HÖLZLE, U., AND SINGH, A. K. Dynamic query-based debugging. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (1999), vol. 1628 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 135–160.

[28] LENCEVICIUS, R., HÖLZLE, U., AND SINGH, A. K. Dynamic query-based debugging of object-oriented programs. *Automated Software Engineering 10*, 1 (2003).

[29] LIU, Y. A. CACHET: An interactive, incremental-attribution-based program transformation system for deriving incremental programs. In *Proceedings of the Knowledge-Based Software Engineering Conference* (1995), pp. 19 – 26.

[30] LIU, Y. A., AND STOLLER, S. D. From recursion to iteration: what are the optimizations? In *Proceedings of the ACM SIGPLAN workshop on Partial Evaluation and Semantics-based Program Manipulation* (New York, NY, USA, 1999), ACM Press, pp. 73–82.

[31] LIU, Y. A., AND STOLLER, S. D. Optimizing Ackermann's function by incrementalization. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation* (New York, NY, USA, 2003), ACM Press, pp. 85–91.

[32] LIU, Y. A., STOLLER, S. D., GORBOVITSKI, M., ROTHAMEL, T., AND LIU, Y. E. Incrementalization across object abstraction. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 2005), ACM Press, pp. 473–486.

[33] LIU, Y. A., STOLLER, S. D., LI, N., AND ROTHAMEL, T. Optimizing aggregate array computations in loops. *ACM Transactions on Programming Languages and Systems 27*, 1 (2005).

[34] LIU, Y. A., STOLLER, S. D., AND TEITELBAUM, T. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems 20*, 3 (1998).

[35] LIU, Y. A., AND TEITELBAUM, T. Caching intermediate results for program improvement. In *Proceedings of the ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation* (New York, NY, USA, 1995), ACM Press, pp. 190–201.

[36] MARTIN, M., LIVSHITS, B., AND LAM, M. S. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 2005), ACM Press, pp. 365–383.

[37] MEIJER, E., BECKMAN, B., AND BIERMAN, G. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2006), ACM Press, pp. 706–706.

[38] MICROSOFT. The C# specification 3.0.

[39] NIEMEYER, P., AND LEUCK, D. BeanShell — Lightweight Scripting for Java. `http://www.beanshell.org/`, 2003.

[40] POTANIN, A. The Fox - a tool for java object graph analysis. Tech. Rep. 02/28, School of Mathematical and Computing Sciences, Victoria University of Wellington, `http://www.mcs.vuw.ac.nz/comp/Publications/`, November 2002.

[41] PRATT, T. W. Control computations and the design of loop structures. *IEEE Transactions on Software Engineering SE-4*, 2 (1978).

[42] RUSSELL, C. Java Data Objects (JDO Specification JSR-12), 2003.

[43] SOLOWAY, E., BONAR, J., AND EHRLICH, K. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM 26*, 11 (1983).

[44] UNGAR, D., AND SMITH, R. B. Self: The power of simplicity. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (New York, NY, USA, 1987), ACM Press, pp. 227–242.

[45] WAAS, F., AND GALINDO-LEGARIA, C. Counting, enumerating, and sampling of execution plans in a cost-based query optimizer. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2000), ACM Press, pp. 499–509.

[46] WATERS, R. C. A method for analyzing loop programs. *IEEE Transactions on Software Engineering SE-5*, 3 (1979).

[47] WATERS, R. C. Cliche-based program editors. *ACM Transactions on Programming Language Systems 16*, 1 (1994).

Appendix A

AspectJ

AspectJ [22] is an extension of Java to provide support for Aspect Oriented Programming (AOP) [23]. AOP is a relatively novel programming technique to support dealing with what are known as 'cross cutting concerns'; facets of the program's design that cannot be neatly encapsulated in objects or classes, but which instead need to be scattered over several different parts of the program. The canonical example is logging; it is often necessary for a program to log what it is doing, and this requires manual insertion of calls to logging methods throughout the program. This insertion is tedious and error-prone.

AOP lets us separate out this logging into an aspect that dictates what to log, and how to log it. The aspect can then be 'woven' into the code. 'Weaving' involves inserting the code specified in the aspect, at the points in the program where the aspect dictates they should be inserted. In this manner the logging code can be consolidated into one part of the program, along with instructions about what needs to be logged. Deactivating logging for certain events that no longer need to be logged requires simply changing the aspect's weaving instructions, instead of manually combing through the source files for these events and removing their logging calls. Requiring new types of events be logged is similarly easy.

In AOP terminology, the points in a program which should be modified

are *Join Points*, and the code with which we modify them is called *Advice*. AspectJ supports the specification of join points by providing syntax to define *pointcuts* (which are named patterns that match one or more join points), and to define the advice that should be executed at that pointcut. All the pointcuts and the advice that acts on them that are used for a particular part of the program are bundled together in an aspect. Aspects in AspectJ are much like classes; they have members and methods, and are declared in their own file. Aspects are by default singletons, and essentially invisible to the rest of the program (some advanced AOP techniques notwithstanding).

AspectJ supports defining pointcuts to match a variety of join point types, including method execution, field access, object construction, exceptions thrown, etc. Pointcuts can specify required details, such as the return type of a method call, or the type of the currently executing object, and so on. When types are specified, appending + indicates that subtypes of that type are acceptable. Pointcuts can also use wildcards, indicated with an asterisk. For example. the following pointcut matches all calls to the method called 'onClick()' of the class Button, or one of its subclasses:

```
pointcut onOnClick():call(boolean Button+.onClick());
```

To use this pointcut, we specify advice within the aspect. Advice is the code to be executed when a pointcut is matched. The advice can be specified to act before, after, or instead of the code that matched the point cut. The following piece of advice causes the value returned from the onClick() call to be printed to the console:

```
after(boolean b) returning(b) : onOnClick(){
System.out.println("onClick returned : " + b);
}
```

The `returning` argument allows the advice to capture the return value of the method. Advice can also capture the target of the method call, as well as the arguments provided to it.

JQL's cache manager makes use of AOP to instrument query evaluations and replace them with cache accesses whenever possible, and to support incremental updating of query caches. Chapter 4 has more details.

Appendix B

HandOpt Code

```
public ArrayList<Object[]> handOptEval(ArrayList<Integer> array1,
                                       ArrayList<Integer> array2,
                                       ArrayList<Integer> array3,
                                       ArrayList<Integer> array4) {
    HashMap<Integer,ArrayList<Integer>> map;
    map = new HashMap<Integer,ArrayList<Integer>>();
    for(Integer i1 : array1) {
        ArrayList<Integer> grp = map.get(i1);
        if(grp == null) {
            grp = new ArrayList<Integer>();
            map.put(i1,grp);
        }
        grp.add(i1);
    }
    ArrayList<Object[]> matches = new ArrayList<Object[]>();
    Collections.sort(array4);
    for(Integer i2 : array2) {
        int b=i2;
        ArrayList<Integer> grp = map.get(i2);
        if(grp != null) {
            for(Integer i1 : grp) {
```

```
                int a=i1;
                for(Integer i3 : array3) {
                    int c=i3;
                    if(b != c) {
                        for(int x=array4.size();x!=0;x=x-1) {
                            int d=array4.get(x-1);
                            if(c < d) {
                                Object[] t = new Object[4];
                                t[0]=a;
                                t[1]=b;
                                t[2]=c;
                                t[3]=d;
                                matches.add(t);
                            } else { break; }
                        }
                    }
                }
            }
        }
    return matches;
}
```